图灵电子与电气工程丛书



# FPGA设计实战

## Design Recipes for FPGAs

[英] Peter Wilson 著
杜生海 等译



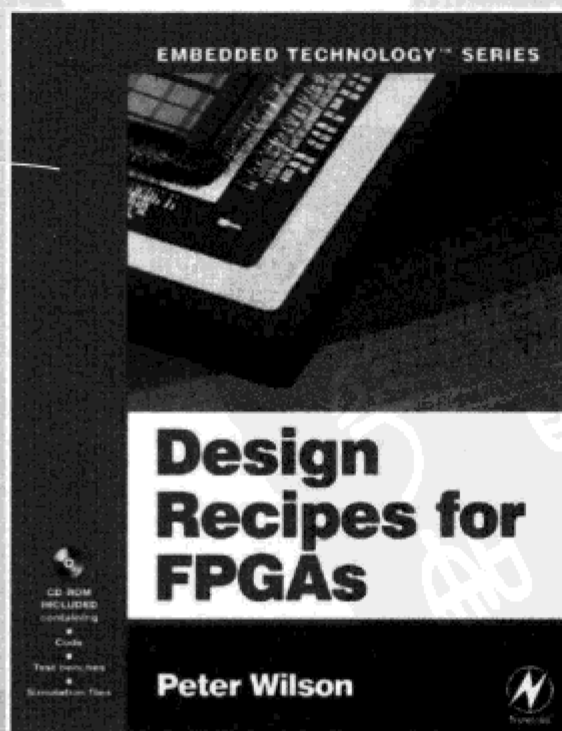

人民邮电出版社
POSTS & TELECOM PRESS

TURING 图灵电子与电气工程丛书

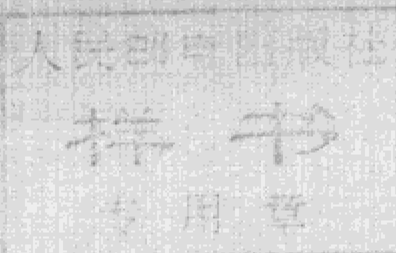
# FPGA设计实战

## Design Recipes for FPGAs

[英] Peter Wilson 著
杜生海 等译




人 民 邮 电 出 版 社
北 京



**图书在版编目（CIP）数据**

FPGA设计实战/（英）威尔逊（Wilson, P. ）著；杜生海等译. —北京：人民邮电出版社，2009.7
（图灵电子与电气工程丛书）
书名原文：Design Recipes for FPGAs
ISBN 978-7-115-20810-1

I. F… II. ①威… ②杜… III. 可编程序逻辑器件－系统设计 IV. TP332.1

中国版本图书馆CIP数据核字（2009）第071214号

## 内 容 提 要

本书是为FPGA工程师量身定制的设计参考指南，不仅介绍了FPGA基本概念，还介绍了设计逻辑和技巧，使读者能够开发出实际高效的代码。

本书适用于电子工程师阅读，也可作为高等院校相关专业师生的参考指南。

图灵电子与电气工程丛书

**FPGA设计实战**

- ◆ 著 [英] Peter Wilson
  译 杜生海 等
  责任编辑 朱 巍
- ◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街14号
  邮编 100061 电子函件 315@ptpress.com.cn
  网址 http://www.ptpress.com.cn
  北京画中画印刷有限公司印刷
- ◆ 开本：787×1092 1/16
  印张：15
  字数：384千字 2009年7月第1版
  印数：1-3 000册 2009年7月北京第1次印刷

著作权合同登记号 图字：01-2009-2899号

ISBN 978-7-115-20810-1/TN

定价：39.00元

**读者服务热线：（010）51095186 印装质量热线：（010）67129223**

**反盗版热线：（010）67171154**

# 版权声明

*Design Recipes for FPGAs* by Peter Wilson, ISBN: 0-7506-6845-3.

Authorized Simplified Chinese translation edition published by the Proprietor.
ISBN: 978-981-272-183-9.


**Elsevier (Singapore) Pte Ltd.**
3 Killiney Road
#08-01 Winsland House I
Singapore 239519
Tel：(65)6349-0200
Fax：(65)6733-1817

First Published 2009
2009年初版

# 致　　谢

首先，我要感谢安德鲁·布朗教授，他是英国南安普敦大学电子与计算机科学学院电子系统设计组（ESD Group）的负责人。他让我有机会进入设计组开始学习，并留下来工作，这才直接导致我能写出这本书。对此，我内心深深地感激他。此外，同事们和ESD研究组的同学们给予了我不断的支持和鼓励，是我坚持下来并不断产生良好想法的不竭动力。

我也要感谢Elsevier出版公司的Tim Pitts，是他帮我开始这项计划，而且在他的鼓励之下我才完成这本书的写作。还要感谢所有为这本书的出版出过力的人，包括Lisa Jones、Helen Eaton、Lewin Edwards、Charon Tec排版公司及其团队成员，还有Elsevier出版公司的所有人员。

最后，衷心地感谢我的家人，尤其是我的妻子Caroline，还有我的孩子Nathan和Heather。无论如何，没有他们的支持，我就不可能完成这本书。

Peter R. Wilson

# 前　　言

本书[1]可以作为那些以现场可编程门阵列（FPGA）为硬件开发平台的工程师、学生和研究开发人员的桌面常备参考书。这本书是仿照*Numerical Recipe*系列丛书的风格编写的，这一系列书每本分别针对不同的编程语言，但目的不是教语言本身，而是教一些必要的方法与技巧，以使应用程序能够工作。本书的基本出发点与此类似，目的是提供一些方法和理解，以使读者能够开发出实际可用的、能够正确运行于FPGA上的VHDL程序。

需要重点强调的一点是，这本书并不是VHDL的语言参考手册。其实，这类手册已经出版了许多，我在书中推荐了一些。这本书有意写成一本VHDL设计参考书，可以看作是对传统VHDL教材的一种补充。



① 本书有附带资源，内容为与设计相关的一些VHDL文件和仿真命令文件，读者可以登录图灵网站（www.turingbook.com）注册后免费获取电子文件。——编者注

# 目　录

## 第一部分　概　述

## 第二部分 应 用



## 第三部分 设计工具箱

第四部分 优化设计

## 第五部分 基本技术

# 第一部分　概　　述

本书共分为五个部分。第一部分为概述，介绍了现场可编程门阵列（FPGA）、VHDL和标准设计流程的基础知识。第二部分介绍了一系列复杂的实际应用的全流程，其中涉及许多关键设计问题，而这些问题是设计人员在工作中经常遇到的。通过这些应用，设计人员可以了解如何由设计规范出发，开发出一套自顶向下的设计方法，并最终构造出详细的设计模块，这些模块可能是以前开发出来的，也可能是由第三方提供的。第三部分则讨论了一些重要的技术方法，都使用了实际的例子进行说明，这样读者可以清楚地了解怎样实现一个特定的功能。这一部分实际上是一个工具箱，其中包含了许多高级的功能实现，而这正是现代数字设计普遍需要的。第四部分讨论了设计优化中的一些重要问题及其高级技巧。例如，怎样使设计出的东西运行得更快或者面积更小等。第五部分详细说明了用VHDL实现基本功能的问题。这部分是针对那些仅有少量VHDL背景的设计者，或许他们正在寻找简单的例子，或者正要解决一个特别具体的问题。

# 第1章　绪　　论

## 为什么用FPGA

设计人员在进行定制化电子设计时，对于硬件平台其实有很多种选择，例如嵌入式处理器、专用集成电路（ASIC）、可编程微处理器（PIC）、现场可编程门阵列（FPGA）以及可编程逻辑器件（PLD）。最终选择哪种技术应该主要取决于设计需求，而不是个人对于某种技术的偏好。

例如，如果某个设计需要一种可编程器件，以便于设计的多次变更，同时算法中又包含了一些乘法和循环这样的复杂操作，那么选择某种专门的信号处理设备（如DSP）就更有意义，因为可以很容易地用C或者其他高级语言进行编程。如果速度要求不是特别严格，而且需要一个小而便宜的硬件平台，那么通用微处理器（如PIC）将是理想的选择。最后，如果要求硬件具有较高的性能，例如工作在数百兆赫兹，那么FPGA将是恰当的选择，因为FPGA不但有较高的性能，同时又具备可编程逻辑的灵活性和可重用性。

3

另外一个要考虑的因素是硬件设计中的优化等级。例如，用C语言编写一个简单的程序，然后对PIC进行编程，其性能可能很有限，因为处理器无法对关键函数进行并行操作。而这在FPGA中却可以用并行化和流水线方式很容易地实现，其处理能力要比PIC高得多。

选择硬件平台的一般原则是：先确定设计需求和硬件选项，然后在考虑这些的基础之上选择一个合适的平台。

例如，如果设计需要一个能达到100MHz的时钟，那么FPGA是不错的选择。如果时钟速率仅有3~4MHz，选择FPGA显然会使成本过高。

如果设计中需要一个灵活的处理器，尽管今天的FPGA已经支持嵌入式处理器，但是用DSP或者PIC却显得更可取。如果设计中需要专门的硬件功能，很明显FPGA是一个合适的选择。

如果设计中需要如乘法和加法等特殊的硬件功能，那么DSP可能是最好的选择，但是如果进行定制化硬件设计，FPGA更加适合。

如果设计需要一些简单而且非常小的硬件模块，PLD或者CPLD（复杂可编程逻辑器件）可能是最好的（简单而小巧的可编程逻辑）。不过如果设计中有乘法操作，或者有复杂的控制器和特殊的硬件功能，FPGA则是最合适的。

做出这种决定还与相关硬件的复杂度有关。例如，VGA控制器或许需要一个FPGA而不是PLD器件，主要是由于硬件的复杂度。另一个有关的因素是灵活性和可编程性。如果使用了FPGA，并且其中的资源没有被用完（如使用了60%），那么若通信协议改变了，或者更新了，FPGA在将来还有足够的空间支持多次的变更或升级。

通过这些简单的指导，就可以对最好的平台做出明智的选择，而且基于这些假设可以选择好具体的器件。大多数综合软件包都有一个好处，那就是让用户在最终选定硬件前可以测试多种设计平台的性能和利用率（如PLD和FPGA等）。

4

# 第2章　FPGA基础知识

## 2.1　引言

本章为不熟悉现场可编程门阵列（Field Programmable Gate Array，FPGA）技术的读者介绍一些基础知识。设计硬件时，了解一些相关的背景知识是有用的，例如硬件描述语言（VHDL）模型的重要性以及与实际设计之间的关系。

## 2.2　FPGA的发展

自从20世纪70年代数字逻辑硬件诞生以来，已经生产出了大量的各类器件，结果无处不在的TTL逻辑系列至今仍在使用（如74/54系列），后来又延续到CMOS技术（如HC、AC、FC、FCT、HCT等）。尽管一直以来（包括今天）印制电路板（PCB）上已经大量使用这些器件，但过去20多年，人们一直在努力为这些基本数字器件引入更强的可编程性。

之所以有这种需求，是由于大多数数字系统使用了两种不同的设计方法。从硬件的角度看，发展动力是提高性能：更快、更小、功耗更低、价格更便宜。这就产生了定制化集成电路设计（专用集成电路，ASIC），每一块芯片都要独立设计、布局、制造和封装。对于大批量的产品而言，这样可以节约成本，但是这种方式显然需要巨额的费用（在当前的硅制程上，做一次掩膜可能需要50万美元），花费的时间也很长（可能一年左右）。



但是，从软件的角度看，更倾向于使用一个标准的处理器架构，例如Intel的奔腾处理器（Pentium）、IBM的PowerPC处理器和ARM公司的ARM处理器等，这样只要开发出应用软件然后下载到这些平台中即可。这种方式显然比搭建一个平台要快得多，但是由于对操作系统的需求、编译器的低效率，以及处理器上的软硬件间的复杂关系导致的性能下降等原因，经常会有一些重大的管理开销。

结果，作为一种折中的方式，可编程器件就被开发出来了。它拥有众多的优点：在高性能的平台上进行硬件设计，拥有最优化的资源，不需要操作系统，可重新配置等。

## 2.3　可编程逻辑器件

第一类可编程器件称为可编程阵列逻辑（Programmable Array Logic，PAL）。这类器件由通过互连阵列连接在一起的逻辑门阵列组成。这些器件内含少量的触发器（通常不超过10个），可以实现小型的状态机（如图2-1所示）。

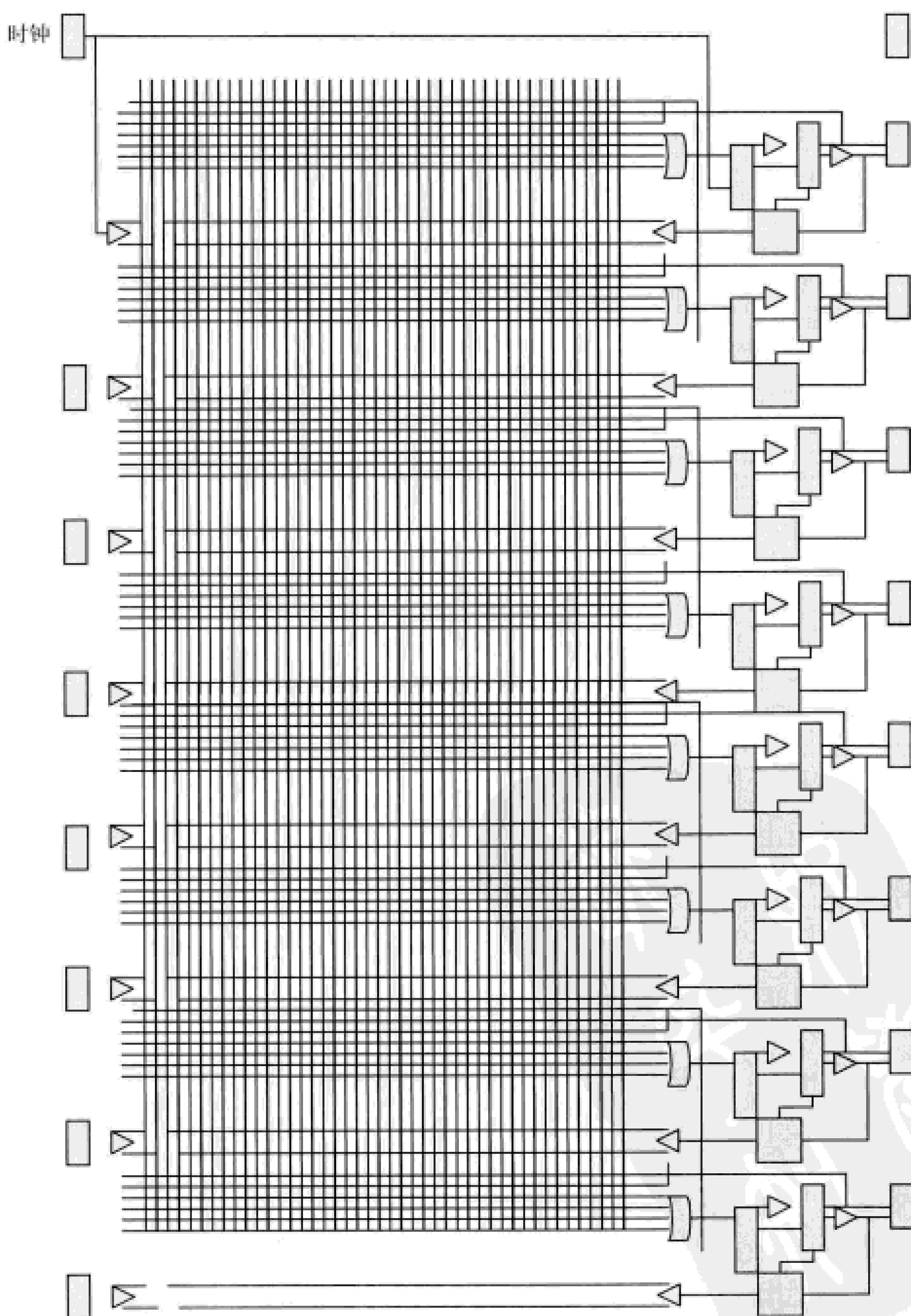



图2-1　可编程逻辑器件

为了突破简单PAL器件的限制，后来开发出了复杂可编程逻辑器件（CPLD）。此类器件的基本原理与PAL相同，只是多了一系列宏块（每一个宏块都相当于一个PAL），这些宏块由布线宏块连接在一起（如图2-2所示）。

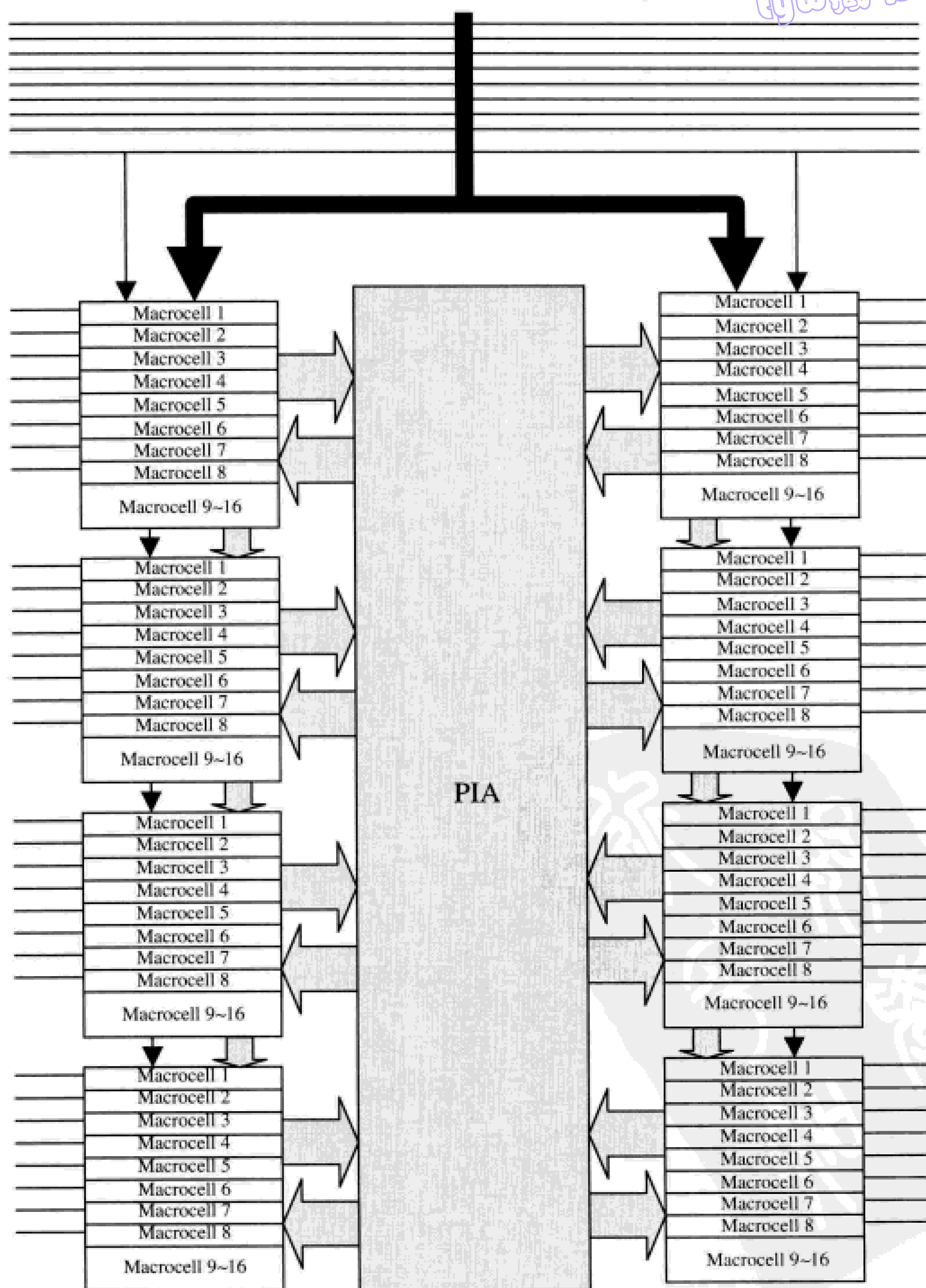



图2-2 复杂可编程逻辑器件

## 2.4　现场可编程门阵列

现场可编程门阵列（FPGA）是CPLD的下一代产品。FPGA没有用固定的门阵列，而是使用了复杂逻辑块（Complex Logic Block，CLB）。CLB是可配置的，不但可以在器件内布线，而且每个逻辑块都能进行优化配置。典型的CLB如图2-3所示。

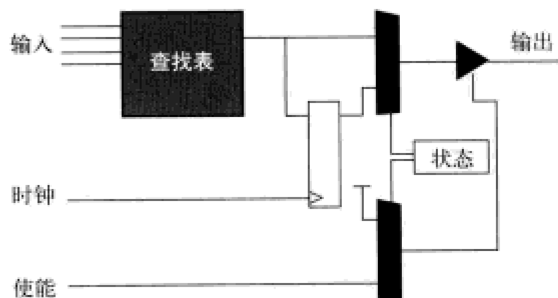



图2-3　FPGA复杂逻辑块（CLB）

CLB中包含一个查找表（Look-Up Table，LUT），可以通过编程的方式将这个查找表配置成某种逻辑功能。CLB内还有一个D触发器，这样CLB就可以配置成组合逻辑（无时钟信号），也可以配置成同步逻辑（带时钟信号），同时还有一个使能信号。Xilinx公司的CLB结构如图2-4所示，从图中可以看出，在实际的器件中有2个4输入查找表、各种多路选择器和2个触发器。

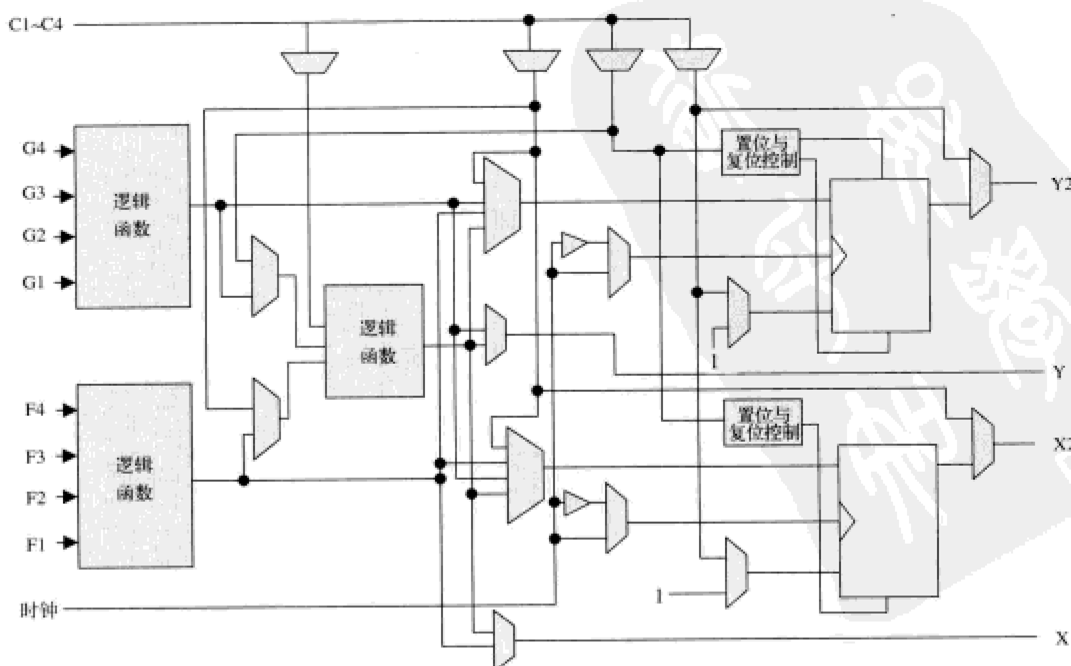



图2-4　Xilinx公司的CLB结构

根据型号的不同，典型的FPGA单个设备内通常有数百到数千个CLB，因此在单颗FPGA芯片内就可以实现非常复杂的器件才能实现的功能，而且很容易配置。现代的FPGA完全有能力在单个设备内容纳多个32位处理器。典型FPGA的CLB布局如图2-5所示。

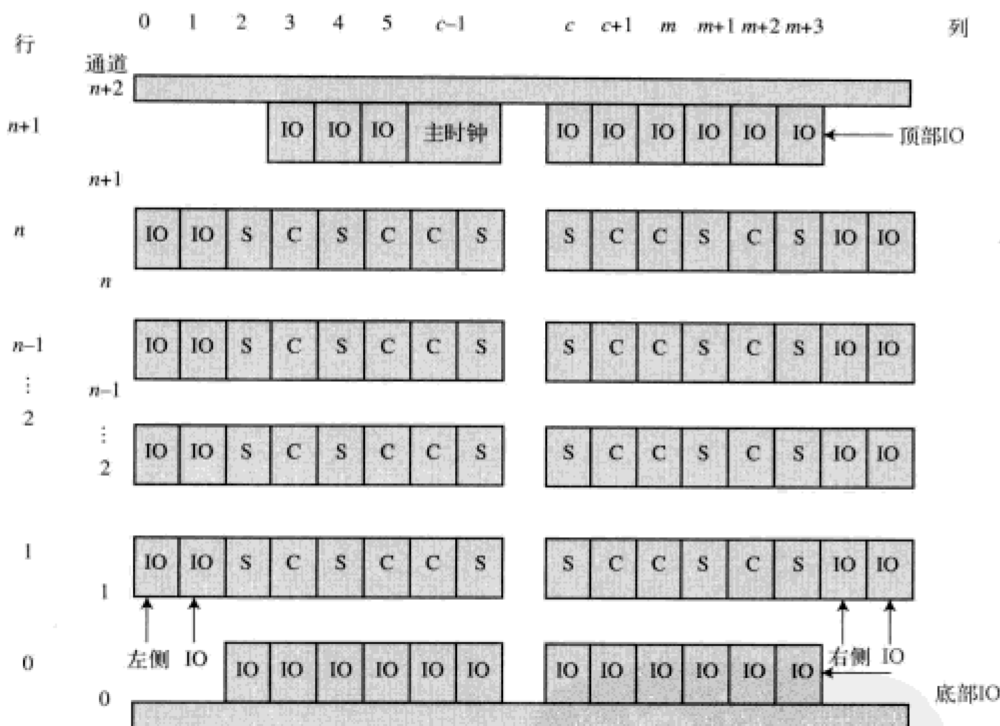



图2-5 FPGA复杂逻辑块（CLB）结构

6~9

## 2.5 FPGA设计技术

当我们用VHDL进行设计时，需要将设计功能映射到FPGA的底层逻辑块上。为做到这一点，需要如下3个步骤。

(1) 映射（mapping）：将逻辑功能映射到CLB上。

(2) 布局（placement）：将CLB放置在FPGA的适当位置。

(3) 布线（routing）：在CLB之间布线连接。

很显然，对于今天的复杂设计而言，“手工”进行设计是不可能的，因此，必须依赖综合软件将VHDL设计转换为可以映射到FPGA CLB上的逻辑功能。这一设计流程是一个反复的不断优化的过程，最终达到一个完整的设计流程。本书后面的章节将详细讨论这一问题。

## 2.6 FPGA的设计约束

如果没有仔细考虑目标FPGA平台，用VHDL很容易写出不切实际的设计。很明显，FPGA的逻辑块和布线资源是有限的，因此设计必须考虑这一点。设计人员使用的VHDL编码风格应该充分利用资源，本书将就如何达到这一目标给出一些例子。VHDL代码可能是独

立于制造工艺的，但是由于这些约束的存在，为了得到最好的结果可能还是需要重写代码。

## 2.7 小结

本章介绍了FPGA的基本技术及其发展历程，重点说明了一些关键设计事项和重要的设计技术。后面的章节将从设计细节或者方法学的角度对此进行详细阐述。

10

# 第3章 VHDL基础

## 3.1 引言

本章并不是一本全面的VHDL参考书，市面上已经有很多优秀的书籍可以作为VHDL的参考书。这里仅仅就VHDL重要的语言结构和用法进行简要的说明和概括，它们对实际应用很有帮助，而且容易使用，但并不一定要很全面地介绍。

本章将介绍VHDL的主要概念和大多数VHDL设计需要用到的重要语法，尤其是与FPGA设计相关的语法。大多数情况下，决定使用VHDL而不是其他语言（如Verilog和SystemC）进行设计，主要不是由于设计人员的选择，而更多地在于可用的开发软件和公司的决定。在刚刚过去的十多年间，VHDL阵营和Verilog阵营之间就两者哪个是最好的语言打起了几近疯狂的“口水仗”，但大多数情况下，这些都完全没有意义，因为项目更关注设计本身而不是使用了什么语法。VHDL和Verilog之间在细节上有很大的差异，但是从历史的角度看，两者最基本的不同还在于两种语言所处的设计环境。Verilog语言根源于传统的“自底向上设计”，被集成电路工业基于单元的设计大量采用，而VHDL语言更多地是为“自顶向下设计”而开发的。当然，这些都是概括性的，而且在现代的设计中也早已经过时了，但是从两种语言的基本语法和编程方式上可以很明显地看到这一点。



撇开Verilog和VHDL之间的细微差别不说，VHDL一个重要的优点是能够使用构建于不同构造体的多层次模型，如图3-1所示。

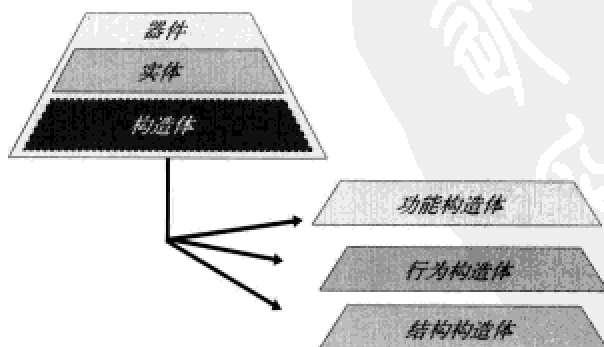



图3-1 具有不同构造体的VHDL模型

这并不是VHDL所独有的，事实上，Verilog语言中也存在一个单独的“模块”有不同行为的概念；但在VHDL中是显式定义的，在将几个实际的多层次设计集成到一起时显得特别有用。将模型分解成接口部分[ VHDL中的“实体”(entity)] 和行为部分 [VHDL中的“构造体”(architecture)] ，这种方式对于在一个单一的接口内建模多种行为是一种极其实用的方法，可以使模型便于交换，直接进行多种实现。

本章其余部分将描述VHDL的一些关键问题，先介绍如何用entity和architecture进行基本

的模型结构定义，然后讨论一些重要的变量类型，再回顾一下并发、串行和分级行为的建模方法，最后介绍VHDL中必需的基本数据类型。

## 3.2 实体：模型接口

### 3.2.1 实体定义

实体（entity）定义了一个VHDL设计模块如何与其他VHDL模块进行连接，同时也定义了模型的名称。实体中还可以定义任何要传递给模型的参数。基本的实体模板如下：

```
entity <name> is
....
entity <name>;
```

12

如果实体的名称为test，那么实体模板将有如下形式：

```
entity test is
end entity test;
```

或者：

```
entity test is
end test;
```

### 3.2.2 端口

使用端口（port）可以将实体连接在一起。在实体中定义端口的方法是：

```
port (
...list of port declarations...
);
```

端口声明定义了连接的类型和连接方向。例如，一个bit型输入端口in1，其端口声明如下：

```
in1 : in bit;
```

如果模型有两个bit型输入端口in1和in2，以及一个bit型输出端口out1，那么端口的声明为：

```
port (
      in1, in2 : in bit;
      out1 : out bit
);
```

由于实体之间的连接点同内部过程之间的连接点一样有效，因此也是很有用的信号，可以在VHDL模型内使用。

### 3.2.3 通用属性语句

如果模型中有参数，可以使用通用属性（generic）语句定义。generic的声明如下：

```
generic (
...list of generic declarations...
```

```
);
```

通用属性的声明与常数定义类似，如下所示：

13

```
param1 : integer := 4;
```

例如，一个模型有两个通用属性：gain（integer）和time_delay（time）。它们在entity中的定义如下：

```
generic (
        gain : integer := 4;
        time_delay : time = 10 ns
);
```

### 3.2.4 常数

实体中也可能包含一些模型专用的常数（constant），可以使用标准的常数声明进行定义，例如：

```
constant : rpullup : real := 1000.0;
```

### 3.2.5 实体举例

为了举例说明一个完整的实体，我们在以下的例子中包含了前面介绍的端口（port）、通用属性（generic）和常数（constant），从而构成一个完整的实体。

```
entity test is
        port (
             in1, in2 : in bit;
             out1 : out bit
        );
        generic (
             gain : integer := 4;
             time_delay : time := 10 ns
        );
        constant : rpullup : real := 1000.0;
end entity test;
```

## 3.3 构造体：模型的行为

### 3.3.1 构造体的基本定义

实体描述了模型的接口和参数，构造体（architecture）则定义了模型的行为。VHDL构造体有多种类型，并且允许为同一个实体定义不同的构造体。对于开发而言，这是很理想的，因为寄存器传输级（RTL）和门级构造体可以同时存在于设计中，测试时使用同样的测试平台（test bench）。

14

构造体的基本声明方法如下：

```
architecture behaviour of test is
..architecture declarations
```

```
begin
...architecture contents
end architecture behaviour;
```

或者

```
architecture behaviour of test is
...architecture declarations
begin
...architecture contents
end behaviour;
```

### 3.3.2 构造体声明

在声明构造体名称之后、开始语句之前，需要声明一些局部信号和变量。例如，如果构造体有两个内部信号sig1和sig2，它们将在模型的声明部分进行声明，如下所示：

```
architecture behaviour of test is
      signal sig1, sig2 : bit;
begin
```

此后，这些信号就可以在构造体语句部分使用了。

### 3.3.3 构造体语句

VHDL构造体可以有各种结构来实现不同的功能。简单的组合表达式使用信号赋值设置新的信号值，例如：

```
out1 <= in1 and in2 after 10 ns;
```

需要注意的是，在实际设计中，after 10 ns这种写法是不可综合的。保证设计可综合性的唯一方法是使设计不分辨延迟或者使用同步设计。VHDL组合逻辑设计会由于工艺库的门级延迟而产生额外的延迟，可能造成短脉冲干扰或者竞争冒险。以下例子说明了一个由多个门组合而成的电路，其中使用了内部信号的声明：

15

```
architecture behavioural of test is
      signal int1, int2 : bit;
begin
      int1 <= in1 and in2;
      int2 <= in3 or in4;
      out1 <= int1 xor int2;
end architecture behavioural;
```

## 3.4 进程：VHDL中的基本功能单元

VHDL中的进程是这样一种机制，通过这种机制，顺序排列的语句能够依照正确的次序执行，多个进程块则是同时执行的。每一个进程由以下几部分组成：敏感列表、声明和语句。基本的进程语法如下：

```
process sensitivity_list is
    ... declaration part
begin
```

```
    ... statement part
end process;
```

敏感列表的作用是：当其中特定的信号值发生变化时，激活进程。例如，通常的设计中都会有一个全局时钟和复位信号来控制进程的行为，如下所示：

```
process (clk, rst) is
begin
      ... process statements
end process;
```

在这个例子中，进程只有在clk或者rst的值发生变化时才被激活。实现同样功能的另一种方式是使用wait语句，进程自动激活一次后，在第二次运行前一直等待两个信号的变化。同样的进程可以写成如下形式：

```
process
begin
      ... process statements
      wait on clk, rst;
end process;
```

16

实际上，wait语句所处的位置并不重要，因为VHDL仿真周期在初始化期间对每一个进程都要执行一遍，所以wait语句可以在进程的开头，也可以在进程的末尾，两种情况的行为是相同的。

在进程的声明部分，信号和变量可以按照前面讨论的方式定义为局部的，例如典型的进程有如下形式：

```
process (a) is
      signal na : bit;
begin
      na <= not a;
end process;
```

进程内使用了局部信号na，在外部信号a（对应于另一个进程）发生变化时被激活。

## 3.5 基本的变量类型和操作符

### 3.5.1 常数

当某个值在整个仿真期间都不发生变化时，该元素的类型就是常数。这通常用来初始化参数或者设置固定的寄存器值，以便于比较。VHDL中，常数可以声明成任何类型，例如：

```
constant a : integer := 1;
constant b : real := 0.123;
constant c : std_logic := '0';
```

### 3.5.2 信号

信号是联系进程与其内部语句之间的桥梁。它们就好像设计中的“连线”一样，将所有设计元素连接在一起。仿真信号的时候，仿真器将依次查看最新的信号值，并检查进程的敏

感列表，看是否有引起进程激活的变化发生。

[17] 可以对信号立即赋值，也可以延迟一定时间后赋值，所以要在未来某个时间安排一次事件。有一点很重要，那就是要认识到信号和顺序执行的一组程序代码（如C程序）是不一样的，信号具有并发性，直到进程下一次激活才会稳定。

下面说明信号的声明和赋值：

```
signal sig1 : integer := 0;
signal sig2 : integer := 1;
sig1 <= 14;
sig1 <= sig2;
sig1 <= sig2 after 10 ns;
```

### 3.5.3 变量

信号定义了进程之间的外部连接，而变量则是进程的内部值。变量只能用于顺序方式，而没有进程之间或进程内部信号的并发特性。变量在进程内部使用，声明方式和使用方法如下：

```
variable var1 : integer := 0;
variable var2 : integer := 1;
var1 := var2;
```

注意，在变量赋值上没有延迟的概念，如果需要稍后赋值，必须使用信号。

### 3.5.4 布尔操作符

VHDL有一组内建的标准布尔操作符，这些无需说明。操作符有and（与）、or（或）、nand（与非）、not（非）、nor（或非）和xor（异或）。这些操作符可以用在BIT、BOOLEAN或者逻辑类型上，举例说明如下：

```
out1 <= in1 and in2;
out2 <= in3 or in4;
out5 <= not in5;
```

### 3.5.5 算术操作符

[18] VHDL内还有一组内建的算术操作符，同样也无需做说明。这些操作符如下表所示。

| 操 作 符 | 描 述 | 举 例 |
|---|---|---|
| + | 加法 | out1 <= in1 + in2; |
| - | 减法 | out1 <= in1 - in2; |
| * | 乘法 | out1 <= in1 * in2; |
| / | 除法 | out1 <= in1/in2; |
| abs | 求绝对值 | absin1 <= abs(in1); |
| mod | 取模 | modin1 <= mod(in1); |
| rem | 取余 | remin1 <= rem(in1); |
| ** | 幂 | out1 <= in1 ** 3; |

### 3.5.6 比较操作符

VHDL有一组内建的比较操作符。操作符有=、/=、<、<=、> 和 >=。这些操作符能够应用于各种数据类型，如下所示：

```
in1 < 1
in1 /= in2
in2 >= 0.4
```

### 3.5.7 移位函数

VHDL有6个内建的逻辑移位函数，如下表所示。

| 操作符 | 描述 | 举例 |
|---|---|---|
| sll | 逻辑左移 | reg <= reg sll 2; |
| srl | 逻辑右移 | reg <= reg srl 2; |
| sla | 算术左移 | reg <= reg sla 2; |
| sra | 算术右移 | reg <= reg sra 2; |
| rol | 循环左移 | reg <= reg rol 2; |
| ror | 循环右移 | reg <= reg ror 2; |

### 3.5.8 拼接

VHDL中的拼接函数XE‘VHDL：concatenation’用符号&表示，用法如下：

```
A <= '1111';
B <= '000';
out1 <= A & B & '1'; -- out1 = '11110001';
```



## 3.6 判断与循环

### 3.6.1 if-then-else语句

对于简单的if语句，其基本语法如下：

```
if (condition) then
      ... statements
end if;
```

其中的条件“condition”是一个布尔表达式，形式为a > b或者a = b。需要注意的是，比较操作符是单个等号“=”，不要与其他编程语言中使用的双等号“==”混淆。例如，如果两个信号相等，那么将输出设为高电平，用VHDL描述如下：

```
if ( a = b ) then
      out1 <= '1';
end if;
```

如果判断需要if和else两个选项，那么语句如下：

```
if (condition) then
```

```
        ... statements
else
        ... statements
end if;
```

这样，我们在前面的例子中增加else语句后，形式如下：

```
if ( a = b ) then
        out1 <= '1';
else
        out1 <= '0';
end if;
```

最后，可以使用一般化的形式实现多个if条件：

```
if (condition1) then
        ... statements
elsif (condition2)
        ... statements
... more elsif conditions & statements
else
        ... statements
end if;
```

20

举例如下：

```
if (a > 10) then
        out1 <= '1';
elsif (a > 5) then
        out1 <= '0';
else
        out1 <= '1';
end if;
```

## 3.6.2 case语句

前面介绍的if语句在定义多个条件时，显得相当简单，不过却有些繁琐，case语句可以避免这些问题，不用在每一种情况下都使用布尔表达式就能实现分支的跳转。这对于状态图的定义或者枚举类型表示的状态间的转换尤其有用。下面是一个case语句的例子：

```
case testvariable is
     when 1 =>
          out1 <= '1';
     when 2 =>
          out2 <= '1';
     when 3 =>
          out3 <= '1';
end case;
```

还可以扩展到某个范围的值，而不只是局限于一个值：

```
case test is
     when 0 to 4 => out1 <= '1';
```

也有可能使用布尔条件和等式。对于默认的情况（也就是说不满足列出的任何条件），可以使用when others：

```
case test is
      when 0 => out1 <= '1';
      when others => out1 <= '0';
end case;
```

### 3.6.3 for语句

VHDL中最基本的循环就是for循环。这是一种执行次数固定的循环。for循环的基本语法如下：

```
for loopvar in start to finish loop
    ... loop statements
end loop;
```



也有可能用递减计数执行循环，这种循环的一般形式为：

```
for loopvar in start downto finish loop
     ... loop statements
end loop;
```

for循环的一个典型应用就是对数组进行初始化，如下所示：

```
signal a : std_logic_vector(7 downto 0);
for i in 0 to 7 loop
       a(i) <= '1';
end loop;
```

### 3.6.4 while循环

while循环的循环次数是在内部决定的，与固定次数的for循环相比，这种循环通常是不可综合的。对于FPGA设计而言，while循环是不能使用的，因为综合软件在编译VHDL模型时会报告错误。

### 3.6.5 exit语句

使用exit语句可以从for循环中完全退出。对于条件已经满足、剩余的循环没有必要进行下去的情况，这是很有用的。exit语句的语法如下：

```
for i in 0 to 7 loop
       if ( i = 4 ) then
             exit;
       endif;
endloop;
```

### 3.6.6 next语句

next语句可以从某一次for循环中退出，这与exit语句稍微有些差异，next语句仅仅退出当前的一次循环，而不是完全退出整个循环，下一次循环继续进行。当条件已经满足、本次

循环的剩余部分没有必要执行时，可以使用这条语句。next语句的例子如下：

22

```
for i in 0 to 7 loop
      if ( i = 4 ) then
           next;
      endif;
endloop;
```

# 3.7 层次化设计

## 3.7.1 函数

要设计一个可以在多个构造体内重用的模型，函数（function）是一种封装行为的简单方法。可以在一个构造体内定义局部函数，也可以在一个包（package，下一节讨论）内定义更通用的函数，但是这一节只讨论定义函数的基本方法。一个简单的函数形式是定义一个含有输入和输出变量的函数头，如下所示：

```
function name (input declarations) return output_type is
      ... variable declarations
begin
      ... function body
end
```

例如，一个函数有两个输入，函数功能为将两个输入相乘，定义如下：

```
function mult (a,b : integer) return integer is
begin
      return a * b;
end;
```

## 3.7.2 包

包（package）是一种在VHDL设计团队内发布类型和函数信息的公共方式。包的基本定义如下：

```
package name is
...package header contents
end package;
package body name is
      ... package body contents
end package body;
```

我们已经看到，包由两部分组成：包头（package header）和包体（package body）。包头用来声明类型和函数，包体则含有函数的具体实现。

例如，函数功能在包体内描述，函数的声明则在包头内。举一个简单的例子，函数执行如下的一种简单逻辑功能：

23

```
and10 = and(a,b,c,d,e,f,g,h,i,j)
```

VHDL函数如下所示：

```
function and10 (a,b,c,d,e,f,g,h,i,j : bit) return bit is
begin
    return a and b and c and d and e and f and g and h and i and j;
end;
```

最终得到的包声明将使用包头内的函数头和包体内的函数体：

```
package new_functions is
function and10 (a,b,c,d,e,f,g,h,i,j : bit) return bit;
end;
package body new_functions is
    function and10 (a,b,c,d,e, f, g, h, i, j : bit) return bit is
  begin
      return a and b and c and d and e \ and f and g and h and i and j;
  end;
end;
```

### 3.7.3 元件

通常在包括行为构造方面，过程（procedure）、函数（function）和包（package）都很有用，随着VHDL在硬件设计中的应用，常常有这样一种需求，就是把设计模块封装成单独的元件，以便在更高的设计层次上引用。VHDL中完成这一任务的方法称为元件（component）。使用元件时必须十分小心，因为从VHDL 1987到VHDL 1993，包含元件的方法发生了根本性的变化，这样做是为了保证语言定义的正确性和一致性。

元件是不用包含以前创建的模型就可以将现有VHDL实体和构造体合并到新的设计中的一种方法。第一步是声明元件，方法同函数的声明相同。例如，如果一个实体称为and4，它有4个bit型输入a、b、c、d和一个bit型输出q，那么元件声明将是以下这种形式：

```
component and4
      port ( a, b, c, d : in bit; q : out bit );
end component;
```



在VHDL模型构造体中就可以用网表形式对这个元件进行实例化：

```
d1 : and4 port map ( a, b, c, d, q );
```

需要注意的是，在这个例子中没有对端口名和VHDL信号进行显式映射，管脚是以元件声明中定义的顺序映射的。如果管脚定义未按顺序进行，就需要显式的端口映射：

```
d1: and4 port map ( a => a, b => b, c => c, d => d, q => q);
```

最后一点要注意的是默认绑定。绑定是指当前库中编译过的构造体与使用的元件之间的连接关系。例如，有可能对不同的实例化元件使用不同的构造体，对某个单独的器件可以使用如下的语句：

```
for d1 : and4 use entity work.and4(behaviour) port map (a,b,c,d,q);
```

若想为所有实例化元件指定某种器件，可以使用以下语句：

```
for all : and4 use entity work.and4 (behaviour) port map (a,b,c,d,q);
```

### 3.7.4 过程

过程（procedure）同函数相似，只不过在参数方面更加灵活一些，因为其参数可以是输入、输出或者双向端口。这与函数相比非常有用，因为函数中通常只有一个输出（尽管这个输出可能是数组），使用过程可以避免为管理返回值而创建记录结构体。虽然过程很有用，但是也应该只用在一些小的特定的功能方面。元件应该用于划分设计，而不是过程，在FPGA设计中尤其如此，因为过程的不当使用可能导致原本非常精简的VHDL描述在设计实现时过于臃肿，而且效率低下。下面是一个简单的过程，描述了一个全加器：

```
procedure full_adder (a,b : in bit; sum, carry : out bit) is
begin
    sum := a xor b;
    carry := a and b;
end;
```

25

需要注意，过程所用的语法与变量（而不是信号）相同，不需要返回语句就可以定义多个输出端口。

## 3.8 调试模型

#### 断言

断言（assertion）用来检查模型中某个条件是否已经满足，在调试模型中，断言是非常有用的。举例如下：

```
assert value <= max_value
       report "Value too large";
assert clock_width >= 100 ns
       report "clock width too small"
       severity failure;
```

## 3.9 基本数据类型

### 3.9.1 基本类型

VHDL定义了一些标准的类型作为内建数据类型：

- bit
- boolean
- bit_vector
- integer
- real

### 3.9.2 数据类型：bit

bit数据类型是一种VHDL内建的简单逻辑类型，该类型只有两个合法的值“0”和“1”。凡是定义成bit类型的元素都可以完成VHDL对该类型赋予的逻辑功能。bit类型的信号和变量

声明如下：

```
signal ina : bit;
variable inb : bit := '0';
ina <= inb and inc;
ind <= '1' after 10 ns;
```

26

### 3.9.3 数据类型：boolean

boolean类型主要用于条件判断，所以if语句的条件测试值就是boolean类型。凡是定义成bit类型的元素都可以完成VHDL对该类型赋予的逻辑功能。boolean类型的信号和变量声明如下：

```
signal test1 : Boolean;
variable test2 : Boolean := FALSE;
```

### 3.9.4 数据类型：integer

VHDL中基本的数字类型就是integer类型，其定义范围是–2 147 483 647到+2 147 483 647。在任何VHDL模型中，整数类型的定义都对综合有明显的意义，特别是有效的比特数，所以使用指定范围内的整数将信号或者变量约束在物理边界之内非常普遍。整数用法的例子如下：

```
signal int1 : integer;
variable int2 : integer := 124;
```

有两种子类型（即基于基本类型的新类型）派生于整数类型，它们都是在实际中使用的整数，只不过定义的数值范围不同。

**1. 整数子类型：自然数**

自然数定义的是所有大于或等于0的整数。实际上，自然数是整数中值较大的一部分：

```
natural values : 0 to integer'high
```

**2. 整数子类型：正整数**

正整数定义的是所有大于或等于1的整数。实际上，正整数是整数中值较大的一部分，但不包括0：

```
positive values : 1 to integer'high
```

### 3.9.5 数据类型：字符型

VHDL中除了数字类型外，还有一整套ASCII字符集。从本质上说，字符与数字之间是不能自动转换的，但是在VHDL标准中（IEEE Std 1076-1993）定义的字符有一个隐含的数字排序。字符可以被单独定义，也可以定义成数组从而创建字符串。处理字符最好的方法是使用枚举类型。

27

### 3.9.6 数据类型：实数

VHDL中使用浮点数来定义实数，预定义的浮点类型称为实数类型。实数类型定义的浮点数范围在–1.0e38到+10e38。对于许多FPGA设计来说，这一点非常重要，因为大多数商业综合软件不支持实数类型，就是因为它们是浮点类型。在实际应用中，有必要使用整数或者

定点数，因为它们可以直接或者简单地综合成硬件。定义实数信号或者变量的例子如下：

```
signal realno : real;
variable realno : real := 123.456;
```

### 3.9.7 数据类型：时间

时间的定义必须使用专门的时间类型。时间类型不但包括时间的值，而且还包括时间的单位（用空格与时间值隔开）。时间类型的基本范围在-2 147 483 647和2 147 483 647之间，时间的基本单位是飞秒（fs）。其他所有时间单位都根源于飞秒，如下所示：

```
ps = 1000 fs;
ns = 1000 ps;
us = 1000 ns;
ms = 1000 us;
min = 60 sec;
hr = 60 min;
```

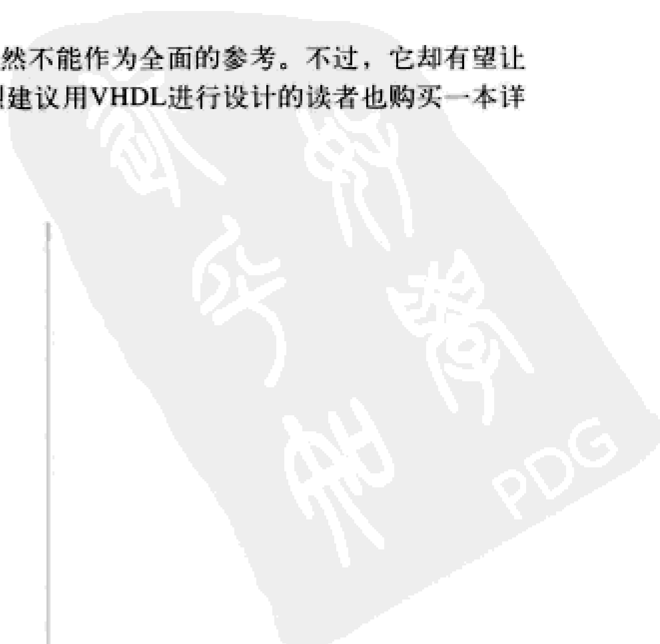
时间定义的例子如下：

```
delay : time := 10 ns;
wait for 20 us;
y <= x after 10 ms;
z <= y after delay;
```

## 3.10 小结

本章仅仅对VHDL做了非常简单的介绍，当然不能作为全面的参考。不过，它却有望让读者有足够的知识理解本书例子中的语法。强烈建议用VHDL进行设计的读者也购买一本详细而全面的VHDL参考书。

28~29



# 第4章 设计自动化与FPGA测试

## 4.1 仿真

### 4.1.1 测试平台

任何硬件设计的总体目标是保证设计满足设计规范的要求。为了衡量这一点，我们不仅需要对硬件描述语言（例如VHDL）所描述的设计进行仿真，而且还要保证无论我们做了什么样的测试，都是合理恰当的，并且能证明设计满足了规范的要求。

设计者用仿真器测试他们的设计所采用的方法是创建一个测试平台（test bench）。测试平台同真实的实验测试平台很类似，都要在输入端施加一些测试激励，然后测试电路的输出响应，看是否满足规范的要求。

实际上，测试平台只是一个简单的VHDL模型，它产生一些必要的测试激励，并检查被测模块的输出响应。通过这种方式，设计者可以观察波形，人工检查结果，或者使用VHDL结构自动检查输出响应。

### 4.1.2 测试平台的目标

任何测试平台的目标都是双重的。首先，保证设计的操作是正确的，这在本质上讲是一种功能测试。其次，要保证综合后的设计仍然满足规范的要求（尤其要重视时序方面的错误）。

30

### 4.1.3 简单的测试平台：实例化元件

以下描述了一个简单的VHDL组合逻辑模型：

```
library ieee;
use ieee.std_logic_1164.all;
entity cct is
     port (   in0, in1 : in std_logic;
              out1 : out std_logic
     );
end;
architecture simple of cct is
begin
     out1 <= in0 AND in1 ;
end;
```

很明显，这个简单的模型是一个两输入与门，要测试这个元件的操作，需要以下几个步骤。

首先，必须在新的VHDL设计中将元件包含进去。然后需要创建一个基本的测试平台。

下面列出了一个基本实体（与外界无连接）的创建过程，构造体内包含了元件声明和测试设计的信号声明。

```
-- library declarations
library ieee;
use ieee.std_logic_1164.all;

-- empty entity declaration
entity test is
end;

-- test bench architecture
architecture testbench of test is
    -- component declaration
    component cct
        port ( in0, in1 : in std_logic;
               out1 : out std_logic
        );
    end component;
    -- test bench signal declarations
    signal in0, in1, out1 : std_logic;
-- architecture body
Begin
    -- declare the Circuit Under Test (CUT)
    CUT : cct port map ( in0, in1, out1 );
end;
```

这个测试平台将在VHDL仿真器中进行编译，不过它却没什么用处，因为在实体中没有定义输入激励（信号in0和in1），而这些激励将驱动被测电路（Circuit Under Test，CUT）。

31

在测试平台中增加激励与一般的VHDL设计相比有一些显著的优点，其中最吸引人的是，在设计测试平台时通常不需要遵循任何设计规则，也不用考虑代码是否可综合。通常测试平台是位于“片外（off chip）”的，因此我们可以按照自己喜欢的风格进行抽象或者行为描述，只要能够达到目的就行。在测试平台中，可以使用wait语句，可以使用文件的读写功能，可以使用断言（assertion），还可以使用其他不可综合的代码选项。

## 4.1.4　增加测试激励

为了在测试平台中增加一组基本的测试激励，我们可以使用简单的信号赋值语句来定义输入信号in0和in1的值：

```
begin
   CUT : cct port map ( in0, in1, out1 );

   in0 <= '0';
   in1 <= '1';
end;
```

显然，这不是非常复杂或者动态的测试平台，我们可以将信号赋值修改一下，让它包含多个数值和定义这一系列值的时间参数，这样就增加了一系列的事件。

```
begin
   CUT : cct port map ( in0, in1, out1 );
```

```
    in0 <= '0' after 0 ns, '1' after 10 ns, '0' after 20 ns;
    in1 <= '0' after 0 ns, '1' after 15 ns, '0' after 25 ns;
end;
```

对于小型的电路来说，这种方法是有效的，但是对于实际中非常复杂的电路，用这种方法产生的值是非常有限的。另一种方法是定义一个常数数组，其中存放由相对简单的测试平台执行的许多组测试激励，对设计施加不同的测试激励，轮流检查设计的输出响应。

例如，我们可以使用“穷尽法”测试前面设计的两输入逻辑，输入数据来自一个record类型结构。VHDL的record类型是一种类型的集合，它将类型分组并构成一种新的类型。

```
type testdata is record
      in0 : std_logic;
      in1 : std_logic;
end;
```

32

使用新的组合类型，例如record，我们可以创建一个数组，就如同其他标准的VHDL数据类型一样。这需要另一种类型声明，即数组类型本身的声明。

```
type data_array is array (natural range <>) of data_array
```

有了这两种新的类型，就可以声明一个data_array类型的常数，实质上是一个testdata类型的含record值的数组，这些值完全描述了用来测试设计的数据。需要注意，data_array类型没有默认的数值范围，但是我们可以在前面描述测试平台中进行定义。

```
constant test_data : data_array := ( ('0', '0'), ('0', '1'), ('1', '0'), ('1', '1') );
```

这种方法的好处是，我们可以从一个需要显式定义每一个测试激励的系统改变到另一个系统，其中通用的测试数据进程只需要从预先定义的数据数组中读取数据即可。这里给出一个简单的测试实例，例子中的进程依次应用了每一组测试数据：

```
process
begin
        for i in test_data'range loop
              in0 <= test_data(i).in0;
              in1 <= test_data(i).in1;
              wait for 100 ns;
        end loop
        wait;
end process;
```

在这个用VHDL写成的测试平台中，有几个有趣的地方。首先，我们可以使用行为级的VHDL语句（如wait for 100 ns），因为不用考虑将代码综合成硬件。其次，通过使用range操作符，测试平台可以不受数据集大小的限制。最后，使用层次结构test_data(i). in0和test_data (i). in1来分别访问单独的record元素。

## 4.2 库

### 4.2.1 引言

VHDL语言本身所具有的数据类型和基本元件模型是非常有限的。因此，为了方便设计

[33] 重用，方便标准数据类型用于模型交流、重用和综合，出现了库的概念。标准VHDL设计中，最主要的库是IEEE库。在IEEE设计自动化标准委员会（Design Automation Standards Committee，DASC）中，各工作组开发出了多种库、包和标准VHDL的扩展。下面列举出其中的一部分：

- IEEE Std 1076 Standard VHDL Language
- IEEE Std 1076.1 Standard VHDL Analog and Mixed-Signal Extensions (VHDL-AMS)
- IEEE Std 1076.1.1 Standard VHDL Analog and Mixed-Signal Extensions – Packages for Multiple Energy Domain Support
- IEEE Std 1076.4 Standard VITAL ASIC (Application Specific Integrated Circuit) Modeling Specification (VITAL)
- IEEE Std 1076.6 Standard for VHDL Register Transfer Level (RTL) Synthesis (SIWG)
- IEEE Std 1076.2 IEEE Standard VHDL Mathematical Packages (math)
- IEEE Std 1076.3 Standard VHDL Synthesis Packages (vhdlsynth)
- IEEE Std 1164 Standard Multivalue Logic System for VHDL Model Interoperability (Std_logic_1164)

这些工作组中，每一个都由来自学术机构、EDA行业和用户社团的志愿者组成，他们精诚合作，共同开发出了IEEE标准（一般每4年修订一次）。

### 4.2.2 库的使用

为了使用库，首先必须声明库：

```
library ieee;
```

每一个库中都定义了许多VHDL包，包中有专门的数据类型或者函数供设计者在设计中调用。例如，在数字系统设计中，我们需要逻辑数据类型，而在基本VHDL标准IEEE-1076中并没有定义这些。标准VHDL只定义了整数类型（integer）、布尔类型（boolean）和比特类型（bit），没有标准逻辑的定义。很明显，这在数字设计中是必需的。出于这个目的，又 [34] 开发出了一个合适的IEEE标准，即IEEE 1164。非常重要的一点是，IEEE 1164标准并不是VHDL标准IEEE 1076的子集，而是专门为硬件描述语言定义的。

### 4.2.3 标准逻辑库

IEEE库中有许多标准逻辑库（std_logic library）可用，如下所示：

- std_logic_1164
- std_logic_arith
- std_logic_unsigned
- std_logic_signed
- std_logic_entities
- std_logic_components
- std_logic_misc
- std_logic_textio

为了在设计中使用某个包的某个元素，用户需要使用USE命令声明对这个包的应用要求。

例如，为了使用标准IEEE逻辑库，需要在库声明之后增加如下的声明：

```
library ieee;
use ieee.std_logic_1164.all;
```

包std_logic_1164对于大多数数字设计非常重要，尤其是FPGA，因为在这个包中定义了所有的商业仿真和综合工具都要使用的标准逻辑类型，此包包含于标准库中。std_logic_1164不仅定义了标准逻辑类型，而且还包含转换函数（标准逻辑类型之间的转换），另外还可以处理有符号数、无符号数和逻辑数组变量之间的转换。

### 4.2.4 std_logic类型定义

std_logic是一种非常重要的类型，这一节专门进行讨论。std_logic类型有如下定义。

- 'U'：未初始化；该信号未经设置。
- 'X'：未知；无法确定其值或其结果。
- '0'：逻辑0。
- '1'：逻辑1。
- 'Z'：高阻。
- 'W'：弱信号，无法确定应该是0还是1。
- 'L'：可能变向0的弱信号。
- 'H'：可能变向1的弱信号。
- '-'：无关项。



通过这些定义，数字设计中可以以一种标准的方式使用逻辑信号，而这种方式在各种软件工具和平台之间都是可预测和可重复的。下面列举出一些对基本std_logic数据类型执行的操作，它们属于VHDL内建的标准逻辑函数：

- and（与）
- nand（与非）
- or（或）
- nor（或非）
- xor（异或）
- xnor（同或）
- not（非）

下面的例子使用std_logic库定义了一个简单的逻辑门——三输入与非门：

```
library ieee;
use ieee.std_logic_1164.all;

entity nand3 is
      port (in0, in1, in2 : in std_logic;
                  out1 : out std_logic );
end;

architecture simple of nand3 is
begin
      out1 <= in0 nand in1 nand in2;
end;
```

# 4.3 综合

## 4.3.1 综合设计流程

36

基本HDL设计流程见图4-1。

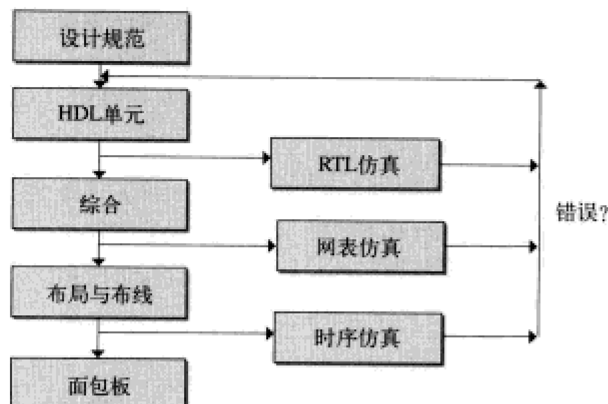



图4-1 HDL设计流程

从图4-1中可以看出，综合是高层次设计与物理布局布线（设计流程的最后阶段）之间非常关键的阶段。综合有几种不同的类型：行为级综合，RTL综合，还有最终的物理综合。

所谓行为级综合，是指将高抽象级的模型综合成一种可以物理实现的中间格式模型。行为级模型可以用无法直接综合的VHDL代码编写，但实际上，在编写高层次模型的时候必须注意，一定要保证这种综合能够执行下去。只有少量的软件工具可以综合行为级的VHDL模型，其中包括Synopsys公司的Behavioral Compiler和英国南安普敦大学的一个研究性综合平台MOODS。

RTL综合就是大多数设计人员常说的综合，是一种结构上的直接映射，寄存器级VHDL设计可以被综合成某种特定FPGA平台上的门级结构。在这一阶段，可以进行详细的时序分析，也可以估计一下功耗。商业的综合软件包有很多种，包括Design Compiler（Synopsys公司），Leonardo Spectrum（Mentor Graphics公司）和Synplify（Synplicity公司）。这里列出的并不全面，实际上有很多种价格不一的各类综合软件。

物理综合是综合设计流程中最后一个阶段。在这个阶段，软件根据特定的FPGA平台将各个单独的门级结构进行布局（用floorplan）与布线。

37

## 4.3.2 综合相关事项

综合过程主要是将类似程序的VHDL代码转换成真实的硬件设计，即网表（netlist）。综合需要有一组输入、VHDL描述、时序约束（即输出需要在何时稳定，输入在何时会稳定，连线的大概延迟，等等）、要映射的工艺（即一些基本模块及其物理尺寸和时序信息）和设计的综合优先权（即面积优先还是速度优先）。

对于大型的设计，通常会将VHDL代码分解成多个模块，然后分别综合。20世纪90年代，每个模块一万门左右是一个比较合理的规模，不过现在的软件工具完全可以处理更大的规模。

### 4.3.3 RTL设计流程

绝大多数标准综合软件需要RTL级的VHDL代码作为输入。VHDL的书写形式必须为包含寄存器、有限状态机（FSM）和组合逻辑（函数）的形式。综合软件将这些模块和函数转换成门级结构和FPGA库中的库单元。图4-2描述了RTL设计流程，这一流程比HDL设计流程更加详细。使用RTL级VHDL设计限制了设计者的范围，因为它将算法设计排除在外，后面我们将会看到。这种方法使设计者不得不在较底层的层面上考虑问题，因而最终的代码有时候可能很冗长，也很复杂。同时，在设计的较早阶段就必须考虑结构的问题，这有一定的限制，也不总是可取的、有帮助的。 38

设计过程是从RTL级VHDL代码开始的：

- 仿真（RTL）——需要开发一个测试平台（VHDL）；
- 综合（RTL）——针对某个标准的FPGA平台；
- 时序仿真（结构级）——仿真以检查时序问题；
- 使用标准工具布局布线（例如Xilinx公司的Design Manager）。

尽管综合软件工具有很多种，例如Leonardo Spectrum和Synplify，不过这些软件一般都使用相似的方法和设计流程。

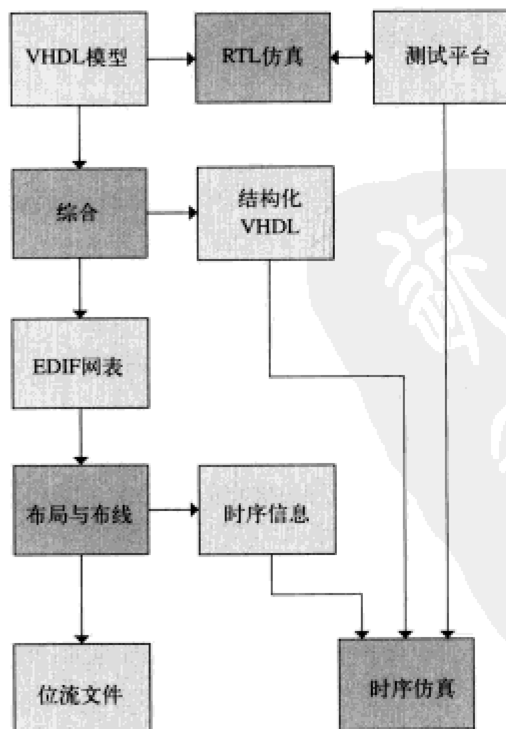



图4-2 RTL综合与设计流程

## 4.4 物理设计流程

综合过程完成后，软件会生成一个网表文件，其中包含了器件和内部互连的信息。布局

布线软件就是根据这个网表文件确定各个模块的具体位置以及它们的互连关系。布局布线后的结果可能并不像想象的那样好。器件和连线资源的利用率通常只有40%~60%。设计者在一定程度上可以牺牲运行时间而换取资源利用率的提高，但是这有极大限制。FPGA供应商一般都会提供一套软件工具（例如Xilinx公司的Design Navigator或Altera公司的Quartus）来管理物理设计中所涉及的各个步骤。

不管选用什么样的物理综合流程，将RTL综合工具输出的VHDL或者EDIF文件转换成一种可以下载到器件中的位流文件，其中有几个步骤基本上是相同的，列举如下：

(1) 翻译（translate）；

(2) 映射（map）；

(3) 布局（place）；

(4) 布线（route）；

39
(5) 生成精确的时序模型和报告；

(6) 产生下载到器件的二进制文件。

## 4.5　布局布线

在当前的商业软件中，主要使用两种技术进行布局布线，它们是递归切割（recursive cut）和模拟退火算法（simulated annealing）。

### 递归切割

递归切割算法中，首先将网表分割成两个大小相等的部分，然后在两个部分之间移动某些模块，使得穿越分割边界的连线数量最少（移动时保持两边的模块数量相等）。这样重复多次，使得区块越来越小。

## 4.6　时序分析

静态时序分析是最常用的一种时序分析方法。这种方法计算所有模块输入到输出的每一条路径的延迟。沿着通过电路的每一条路径将延迟叠加起来，最终得到通过整个设计的关键路径，从而加快了设计的速度。

只要电路中没有环路，这种分析方法就能够很好地工作。但是在某些情况下，分析会变得困难一些。一些设计软件可以帮助设计者在寄存器上打断这些环路并对反馈进行处理。

在时序分析中，设计者都可以牺牲一些精确性而减少运行时间。数字电路仿真软件（如Modelism和Verilog），由于使用了近似的时序模型，从而能够较快地得到分析结果。但是模拟电路仿真软件（如SPICE），由于使用了更为精确的数字，运行时间更长。

## 4.7　设计缺陷

缺乏经验的设计者最常犯的错误是将简单的东西变得很复杂。其实，要完成一个成功的设计，最好的方法是让设计元素尽量简单，管理设计的最容易的方法是有效地使用层次结构。

同设计复杂度直接相关的第二个常犯的错误是没有进行充分的测试。保证设计的所有方面都得到充分的测试是极其重要的。这就是说，不仅要进行基本的功能测试，而且还要进行

系统级的测试，同时要检查多余的状态和可能出现的错误状态。

另一个较为普遍的错误是不必要地使用了多个时钟。多时钟可能产生时序相关的bug，如瞬时脉冲或者对硬件的依赖性。这些问题可能存在于硬件中，但是仿真不一定能验证出来。 40

## 4.8 FPGA设计中的VHDL问题

### 4.8.1 初始化

信号和变量的默认值在综合时会被忽略，因此，设计者必须对所有的触发器进行同步或异步的置位/复位，以便有一个稳定的起始条件。一定要记住，综合工具基本上是很“愚蠢”的，它只能按照一些基本的规则工作，不一定总是能够综合出设计者期望的硬件电路。

### 4.8.2 浮点数及其操作

综合工具目前尚不支持浮点数据类型。浮点数通常需要32位，对于大多数FPGA和ASIC平台，硬件需求太大了。

## 4.9 小结

本章介绍了开发测试平台（test bench）和仿真验证VHDL模型所遇到的一些实际问题。这在VHDL（或其他硬件描述语言）中是经常被忽略的技巧，而且最重要的是要确保最终实现的设计具有正确的功能。另外，也介绍了设计综合的概念，重点强调了如下问题：既要保证设计仿真的正确性，又要保证设计能够被正确地综合到目标工艺上，而且带有实际延时和寄生延时也能够正确地工作。最后，提出了一些具体实现时常遇到的问题和实际硬件设计中可能出现的问题，这些问题将在本书第四部分详细讨论。

这里提一个很有用的重要概念，就是确认（validation）和验证（verification）的区别。这两个术语常常被混淆，导致在最终的设计中和满足规范方面出现了问题。确认要保证的是设计“做了正确的事情”。如果规范要求设计一个低通滤波器，那么我们就必须实现一个有效的低通滤波器设计。我们甚至可以更详细地规定，设计必须在某种约束下工作。另一方面，验证相比确认就更细节化一些，可以描述成“正确地做了正确的事情”。换句话说，验证不但要保证设计满足了功能上的要求，而且还要保证设计必须符合规范中定义的所有标准，还要留有一定的余量以保证设计在各种可能的操作条件下都能满足规范的要求。 41~42



# 第二部分 应 用

本书第二部分讨论与应用相关的问题，目的是为设计者指出实际应用中的关键点和各种信息中的一些“精髓”。本书后面部分提到的一些技术信息可供读者参考，使读者仿佛看到一片“森林”，当他们要解决一些具体的问题时，可以从中选择某些“树木”。每个应用都使用了方块图、状态图和代码片段来解释关键的概念。如有必要，设计的某些特定细节将来再进一步讨论。

第一个应用是高速视频监视系统，需要连接一个摄像头，还包括随机存取存储器（RAM）的接口和一个硬盘。这虽然是一个概念性的系统，但却广泛应用于各类行业中。其中所涉及的技术在类似的测试与监控应用中非常有用。

第二个应用更加强调处理能力，举例说明了在标准FPGA平台上开发多处理器内核的实际问题，以及如何去管理。





# 第5章　图像与高速处理

## 5.1　引言

这个应用主要是用来说明如何在FPGA上用VHDL对几个高速数据率的设备进行处理。系统包含一个高速摄像头、一个处理器内核、一个硬盘接口、一个随机存取存储器（RAM）接口和外部程序计数器的串行连接。选择这样一个系统的目的主要是为了说明如何快速而有效地移动大量的数据。图5-1显示了这样一个应用的概况。正如我们所见，有几个关键的地方，但是最主要的还是大量数据的高速、有效且可靠地移动。

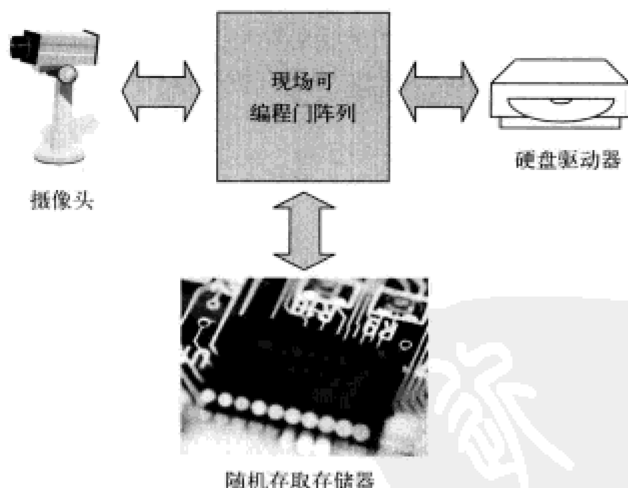





图5-1　视频监视系统略图

该系统影响性能的关键点主要在于3个接口，分别是：

(1) 摄像头和FPGA的接口

(2) FPGA和PC/硬盘驱动器（HDD）的接口

(3) FPGA和RAM的接口

如果我们要考虑摄像头的性能衡量标准，主要有以下4个方面：

(1) 分辨率（resolution）

(2) 帧速率（frame rate）

(3) 色彩规范（color specification）

(4) 图像剪切大小（clip size）

在这个例子中，分辨率定义为640×480像素，色彩模式为24位色彩（3×8位平面），最大的帧速率为100s，基本的图像剪切大小最大10s。

上面的概略图中并没有显示出对一些基本的控制选项［如“播放（play）”、“记录（record）”、“存储（store）”］的需求，而通过这些选项可以用标准的VGA输出接口（大多

数FPGA开发套件中都有这个接口）重放存储的图像，或者将长时间的视频数据存储到硬盘中（或者类似的大容量存储设备中）。这可以通过使用PC接口来单独处理，不过具体的细节超出了本例描述的基本系统的范围。

## 5.2 摄像头接口

### 5.2.1 硬件接口

有很多种方法可以实现摄像头的高速数据传输，其中最常用的两种是通用串行接口（USB）连接到PC，以及使用低电压差摆（Low Voltage Differential Swing，LVDS）串行数据传输的标准摄像头接口。LVDS系统是一个差分串行链路，使用350mV电压实现高速数据的低噪声、低功耗传输。许多FPGA开发套件中都有一条标准的LVDS总线可用，因此可以将摄像头和FPGA板卡之间的信号直接连起来，从而实现摄像头到FPGA的数据传输，然后进行存储（存储到RAM或者硬盘中）。

46

### 5.2.2 数据率

实际的数据率理论上等于分辨率乘以帧速率，再乘以每像素的比特数，可以用以下公式计算：

$$数据率 = 分辨率 \times 帧速率 \times 每像素的比特数 \tag{5-1}$$

对于本章中的例子，数据率如下：

$$数据率 = 640 \times 480 \times 100 \times 24 \tag{5-2}$$

$$数据率 = 737\ 280\ 000\ \text{bit/s} \tag{5-3}$$

这一数据率相当于超过90MB/s，这对实际的应用来说是一个非常快的速率。即使FPGA运行在100MHz，这样一个系统的带宽富余也是少得可怜。

### 5.2.3 拜尔模式

幸运的是，实际的应用中，大多数摄像头系统不会使用这种原始的24比特模式。柯达公司（Kodak）开发出了拜尔（Bayer）模式。使用这种技术，每个像素就不需要用独立的三个色彩平面（共需要24比特）来表示。在图像传感器上面覆盖一个色彩滤波器阵列，这样每像素需要的比特数就限制在了8比特（因为当前像素使用了某种色彩的滤波器）。将拜尔模式以一种固定的顺序重复应用于整幅图像，这个过程可以做标准化处理。拜尔模式如图5-2所示。

很明显，使用这种方法，需要的数据率只是原来的三分之一，降低到了30MB/s，这处理起来就容易得多。

47

但是，这种方法的缺点是分辨率降低了，不过大多数图像可以使用插值的方法很好地恢复出来。插值的方法是，首先检查当前像素是什么颜色（红，绿，蓝），然后分别用相邻像素的其他两种颜色的平均值作为当前像素的这两种颜色。例如，如果当前像素的颜色是绿色，那么当前像素的蓝色和红色分别由相邻像素的蓝色（2）和红色（2）平均得到。

### 5.2.4 存储器需求

尽管采用拜尔模式后，需要的绝对数据量减少了，但是对RAM的需求仍然很高，例如

对640×480像素的图像，需要的存储器大小为：

存储器大小 = 分辨率 × 每像素比特数

存储器大小 = 分辨率 × 8 bit

存储器大小 = 640 × 480 × 8 bit

存储器大小 = 307 200 × 8 bit（每帧）

很明显，这需要很大的存储空间，在FPGA内部存储是不太可能的。更切合实际的方案是在FPGA外部连接一些大的存储器（开发板上可能已经有了这些存储器）。对存储器的选择可以是同步动态随机存取存储器（SDRAM），也可以是Flash存储器。本书后面的章节将详细讨论这些类型的存储器，不过在这里先了解一下每种类型的优缺点有助于我们理解本章的内容。

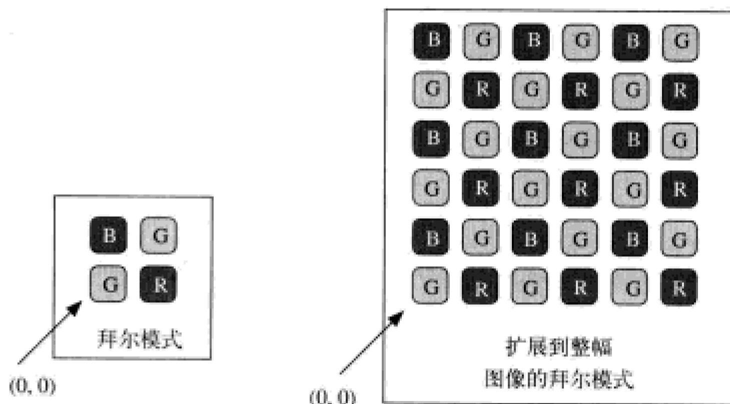



图5-2　基本拜尔模式及其在更大图像区域内的扩展

SDRAM的主要特征如下。

- DRAM依靠逻辑门上的晶体管电容来存储数据。
- DRAM比SRAM（静态随机存取存储器）更小，更紧凑。
- DRAM无法进行综合，必须使用一个单独的DRAM芯片。
- SDRAM需要一个同步时钟，这个时钟要与硬件系统（同微处理器一同进行操作）其他部分的时钟具有一致性。
- DRAM中保存的数据必须定时刷新，因为存储的电荷有衰减现象。
- DRAM的速度要慢于SRAM。

48

静态RAM（SRAM）的工作方式与只读存储器（ROM）类似，也具有一些重要的特征需要注意。

- 存储器的存储单元基于标准的锁存器。
- SRAM速度更快。
- SRAM比DRAM（或者SDRAM）更大。
- SRAM可以被综合到FPGA内，所以对于小规模、快速的寄存器或存储器模块非常理想。

静态RAM本质上是异步的，但是也可以修改为同步的（就像SDRAM就是DRAM的同步版本一样），并且常被称为同步RAM。

从这一点来看，Flash存储器是有用的，即使它的操作与前面介绍的存储器完全不同，

因为Flash存储器使用起来非常容易，而且在FPGA开发板上的使用很普遍。

Flash存储器本质上是一种EEPROM（电可编程只读存储器），可以作为一种耐久性的存储器使用。为什么是耐久性的呢？因为在Flash存储器中，即使断掉电源，存储器中存储的数据也不会丢失，所以常被作为一种只读存储器来使用，在FPGA系统中，Flash存储器常被用来存储FPGA程序，不过也可以用作随机存储器存储当前的数据。

## 5.3 开始

现在，设计的前后关系已经描述清楚了，基本的规范也建立起来了，实际设计中的第一阶段可以开始了。实际上，许多独立的模块可能已经以某种形式存在了，只是可能需要修改以满足特定应用的需求。但是，一般来讲，用自顶向下的设计方法作为开始是比较明智的做法。

所谓自顶向下的设计方法是指，以设计规范为基础，首先设计顶层模块，并给出正确的输入输出接口（在细化设计的时候也有可能改变这些接口），同时也要有一个大概的模块结构，其中包括设计中的功能模块。如果我们以本章中的设计实例作为说明，那么典型的开始点将是一幅顶层设计图，其中显示了各模块间的连接关系以及整个设计的接口。这一阶段不会完成一些细节性的设计，但是却可以构造一个顶层设计，以后设计每一个模块时再往其中“填充”细节。

49

图5-3是本章例子的顶层设计草图。

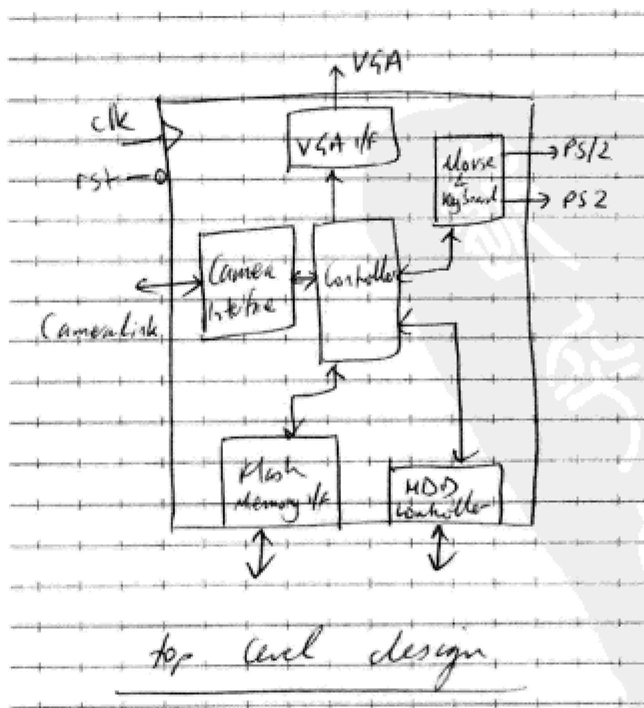



图5-3 顶层设计草图

草图中描述了设计的基本特征：主功能块，主要的接口。同时也要注意，我们增加了一个系统时钟和系统复位，它们连接到了所有独立的功能模块。

另外，在原始的设计中并没有指定用户输入的方式（例如，用户怎样控制摄像头的接口，或者怎样存储数据）。这里决定使用简单的鼠标和键盘接口来为用户提供FPGA系统的控制。[50] 这是一种灵活的方式，首先，我们可以使用鼠标或者键盘上的某些键来初始化一个记录序列、回放或者存储等，但最后也有可能设计一个简单的带有按钮的用户接口，或者具有类似特征的接口，这些接口显示在屏幕上可以控制系统，具体用哪种取决于设计的复杂度。

## 5.4 确定接口

从图5-3所示的草图中，就可以确定顶层设计的接口需求。首先，需要一个时钟信号和一个复位（低电平有效）信号，为了让事情尽量简单（这永远都是一个良好的策略），可以将时钟管脚定义为clk，将复位管脚定义为nrst。这些都是标准的逻辑连接，所以使用IEEE std_logic库中定义的基本的标准逻辑类型。这里并不会定义管脚具体实现时的任何细节，例如管脚是5V、3.3V还是1V，但是却简化了模型中的逻辑级数。具体的实现问题由所用的FPGA定义。

## 5.5 定义顶层设计

对于这个设计，必须定义一个顶层实体名和一个单独的模块名。使用有意义的名称永远都是一个好主意（除非名称很长、很难管理，这种情况下可以用首字母缩写代替），层次关系也有助于减少名称重复的问题。例如，如果设计是一个图像处理器及其存储接口（an image handler and storage interface），那么可以将其缩写为IHSI（记住，VHDL不区分大小写）。顶层模块以下的每一个主要模块都具有前缀为ihsi_的名称。这样做还有一个好处，在编译好的库中，按字母顺序排列后，所有子模块都在一起，方便查找。因此，我们就得到了这个完整应用的第一个顶层实体：

```
library ieee;
use ieee.std_logic_1164.all;
entity ihsi is
    Port (
            clk  : IN std_logic;
            nrst : IN std_logic
    );
end entity ihsi;
```

[51]

然后，我们可以确定哪些主要模块需要外部接口，并在顶层实体中增加必需的连接点。值得记住的是，在设计的每一个阶段，我们都没有必要为了测试设计中的其他部分而完整地定义每一个模块。我们可以使用行为级的模型，甚至是空的模型来简化这一过程，只要保证接口正确即可，以后再用完整的功能模型来代替所有的空模块。开始时，我们可以使用行为级模型，然后用寄存器传输级（RTL）模型代替，最后再用综合后的模型代替。这样，一个完整的系统就可以逐个模块进行测试，直到所有模块都没有问题。

## 5.6 系统模块定义与接口

### 5.6.1 系统分解

在本章给出的特定应用中，有几个重要的带外部接口的模块：

- 鼠标控制器（PS/2）；
- 键盘控制器（PS/2）；
- Flash存储器；
- VGA输出；
- 摄像头连接；
- PC接口。

下面依次对这些接口进行说明，并指定它们在系统中的连接关系。

### 5.6.2 鼠标和键盘接口

鼠标和键盘都是PS/2接口，这相对容易一些。每个接口都有一个时钟和一个数据连接，因此可以定义如下的两个管脚。

鼠标：mouse_clk，mouse_data。

键盘：key_clk，key_data。

一般情况下，PS/2接口（本书第11章和第12章将详细说明PS/2接口）允许双向使用（例如，设备到控制器或者相反），所以在顶层实体中，这些连接必须定义为INOUT类型的std_logic连接。 52

### 5.6.3 存储器接口

对于存储器接口，我们有两种选择。第一种是精确地定义在应用中要使用的存储器类型，如RAM、Flash、EEROM、DRAM和SRAM等，然后制定一个只能与选定类型存储器配合的接口。另一种选择是，在内部将任何类型的存储器都看作通用的RAM，然后定义一个存储器模块同实际的存储器进行接口（例如将存储器接口看作一个虚拟的RAM模块）。因此，在最初的设计中，我们可以将存储器看作一个简单的同步RAM模块，接口具有时钟、数据总线、地址总线和读写信号。对于这个初步的接口，我们只需要如下的信号：

| 信 号 | 名 称 | 方 向 | 类 型 | 备 注 |
|---|---|---|---|---|
| Clock | mem_clk | OUT | std_logic | |
| Data bus | mem_data(31:0) | INOUT | std_logic | |
| Address bus | mem_addr(31:0) | OUT | std_logic | |
| Write | mem_nwr | OUT | std_logic | (active low) |
| Read | mem_nrd | OUT | std_logic | (active low) |

关于存储器接口以及存储器本身的详细描述将在后面有关存储器的章节中给出。

### 5.6.4 显示接口：VGA

对于VGA输出（后面的章节将详细讨论），我们需要为开发板或者系统上的VGA连接器定义一些专门的管脚。任何VGA系统首先需要的是时钟和同步信号。

VGA全局时钟信号需要设定到某个频率（具体取决于显示器），例如25MHz，这必须用FPGA板卡上的系统时钟（100MHz）分频得到。VGA时钟管脚也称为像素时钟（pixel clock），根据命名惯例，我们可以使用vga_作为前缀，后面跟功能名称。所以，对于像素时钟，管脚

53 可以命名为vga_out_pixel_clock。

除了时钟之外，还有3个同步信号是必需的，分别是水平同步（horizontal sync，命名为vga_hsync）、垂直同步（vertical sync，命名为vga_vsync）和复合同步（composite sync，命名为vga_comp_sync）。最后，还有一个消隐脉冲（blank pulse，命名为vga_out_blank_z）。

接下来定义的一组管脚是3种色彩数据。VGA有3个色彩平面（红、绿、蓝），每一个平面由8位数据表示，总共需要24位数据。前面已经讨论过，这一过程可以使用拜尔模式实现，但是当最终的输出像素数据合在一起的时候，由于所有的3个平面都需要输出数据（即使数据都是0），因此可以定义成如下的8位端口：

```
vga_out_red : OUT std_logic_vector (7 downto 0);
vga_out_green : OUT std_logic_vector (7 downto 0);
vga_out_blue : OUT std_logic_vector (7 downto 0);
```

这就是系统到显示器的完整的VGA接口定义。关于VGA接口的更详细的描述将在本书第13章中给出。

## 5.7 摄像头连接接口

摄像头接口标准提供了一个通用的26个管脚的接口，满足大多数摄像头的要求，因此我们只要在顶层设计中定义一个标准接口即可。尽管标准接口需要26个管脚，但实际上是为了满足不同的需求而进行不同的配置，我们的应用中，只要使用基本的接口功能即可，仅仅需要11个管脚。

对于时钟管脚，我们可以定义为camera_clk，4根摄像头的控制线分别为cc1、cc2、cc3和cc4，再加上前缀camera_，因此可以命名为cam-era_cc1、camera_cc2、camera_cc3和camera_cc4。还有两根串行通信线serTFG（连接到图像采集卡）和serTC（连接到摄像头），我们可以将它们分别命名为camera_sertfg和camera_sertc。最后，还有4根来自摄像头的数据线，分别命名为camera_x0、camera_x1、camera_x2和camera_x3。

很明显，实际的接口中需要进行差分输出，因此最终的接口需要将这里定义的简单接口 54 形式转换为连接器中对应的引脚。

## 5.8 PC接口

PC接口可以使用标准的串行接口，例如USB（详见7.6节），也可以使用硬盘驱动器（HDD）的直连接口。

HDD接口对前面讨论过的RAM接口提出了各种挑战。硬盘接口有很多个标准，当前主流使用的主要有两个，分别是IDE/AT（Intelligent Drive Electronics/Advanced Technology）和SCSI（Small Computers System Interface）。SCSI接口普遍应用于高速设备中，历史上被UNIX系统广泛使用。SCSI实际上是一个通用的系统接口，因此几乎允许任何类型的设备连接到SCSI系统总线上。IDE/AT标准是专门为硬盘而设计的，所以具有一些硬盘接口方面的优势。IDE设备一般是慢速设备，但是却比SCSI设备要便宜，因此PC更倾向于使用IDE/ATA接口，更高端的工作站则使用SCSI设备。

在这样的背景下，IDE/ATA设备显得更适合，因为其接口比SCSI接口要简单得多，因此在原型系统开发中常被使用。如果需要设计一个更高级的系统，以后还可以更改接口。IDE

接口总线上可以连接多个主设备和多个从设备（任何人从PC端向外看，都会认可这样一个要求，即安装一块新的硬盘前必须设置一下主/从开关或者跳线）。总线控制器使用命令对一系列寄存器进行设置，然后连接链上被选中的设备即开始工作。值得注意的是，总线将以连接链上最慢速设备的速度工作。

IDE/ATA设备的配置总共有13个寄存器，这些寄存器分为命令块寄存器（command block register）和控制块寄存器（con-trol block register）。命令块寄存器用来对设备发送命令或者查询设备的状态。控制块寄存器用于设备的控制和备用状态的查询。IDE/ATA设备的详细接口情况超出了本书的范围，并且在本例中也没有用到，因此不做进一步讨论。

IDE/ATA接口非常复杂，完全实现可能需要数千行的VHDL代码。如果有这样的性能需求，并且是必须的，读者可以查找大量的源代码信息来实现这一设计，包括ATA 6/UDMA100规范。 55

另一种替代的方法是使用标准接口，例如USB，使用存储器缓存并且对数据进行压缩以方便管理。关于USB接口的详细讨论放在本书第7章中。

## 5.9 小结

本章描述了如何将一个高层次的设计规范分解为一系列在实际中容易解决的问题，这些问题可能都有比较简单的解决方法。一个成功的系统设计，其关键是将设计分解成多个模块，每个模块都有一个可定义的核心功能。这可以用VHDL直接进行实现。另一个方面是分析各个模块的边界。

系统设计者所构造出来的一个共同的短语是“问题迁移到了边界”。换句话说，我们如果了解核心功能就可以很容易地得到VHDL设计，但是，要让这些独立的模块成功地进行通信就困难得多。结果，设计者往往花费大量的调试时间来将不同的功能模块集成在一起，为此还不得不重写许多代码。

处理这种问题的一个有用的方法是，创建一个“空”的VHDL模型，该模型并不进行功能性的操作，而只有正确的接口。可以使用基本的通信测试数据对这些模型进行测试，以保证接口信号的正确性，数据能够以要求的数据率在整个设计中传输。在开发核心VHDL代码之前，必须能够发现信号名称、方向和类型中的错误。

希望本章为读者提供一个用VHDL进行复杂系统建模与设计的有用的介绍，同时也为读者重点说明了一种通用的高层次思考方法，而不用深入考虑每个模块的细节问题。 56

# 第6章 嵌入式处理器

## 6.1 引言

本章通过具体的例子说明如何让FPGA设计与处理器结合起来的一些关键问题。不管是简单的8位微处理器，还是大型的处理器IP核，都需要一个软硬件协同设计环境。本章将带领读者了解如何实现一个行为级的微处理器用于算法评估，如何设计一个可以在FPGA中综合与实现的结构正确的模型。

21世纪，硬件设计人员面临的主要挑战之一就是硬件与软件协同设计的问题。从基于标准硬件架构的软硬件划分机制，到当前的按照性能与功耗要求在编译层次对算法进行优化，适当地用硬件或软件在不同层次上实现，协同设计的概念已经有了改变。

这方面非常适合FPGA，因为它们可以作为固定的硬件架构来运行编译后的软件，也可以实现优化的硬件，比等效的软件具有更高的速度，另外还是一种可配置硬件，能够适应各种环境提出的不断改变的需求。

## 6.2 一个简单的嵌入式处理器

### 6.2.1 嵌入式处理器架构

57

这里举一个嵌入式处理器的应用实例，就是在FPGA平台上实现一个通用的微控制器。图6-1是一个简单的8位微控制器。




图6-1 简单的微控制器

从图6-1中可以看出，微控制器是一个"通用的微处理器"，具有简单的时钟信号（clk）和复位信号（clr），还有3个8位的输入输出端口（A、B和C）。在微控制器内部，需要以下几个基本元件。

(1) 控制单元。用来管理处理器的时钟和复位，管理数据流和指令流，控制端口通信，等等。同时还需要一个程序计数器（PC）。

(2) 算术逻辑单元（ALU）。需要一个至少能够执行一些基本操作的可编程微处理器（PIC）（在ALU中执行）。

(3) 地址总线。

(4) 数据总线。

(5) 内部寄存器。

(6) 指令译码器。

(7) 存放程序的只读存储器（ROM）。

用一个标准的FPGA就可以简单地实现这些独立的单元（1~6），但是ROM实现起来有些困难。如果使用一组寄存器来实现ROM，很显然这在FPGA的结构中是极其没有效率的。然而，大多数现代的FPGA平台中都有一些RAM块，可以将这些RAM块用作ROM，在系统复位时，用ROM的数据对其进行初始化，然后就可以运行程序了。 58

嵌入式内核在这方面存在较为严重的问题，那就是相比专用的处理器内核，嵌入式内核的效率较低。通常会有一个折中，这种情况下，需要以一种不同的方式来实现ROM，但在硬件上有一定的损失。第二个问题是，内核使用什么类型的存储器？在FPGA中，存储器通常可以配置成各种深度（即需要的存储器地址数量）和宽度（数据总线宽度）。例如，一个具有512个地址的8位数据宽度的RAM块等价于一个256个地址的16位数据宽度的RAM块。

如果我们需要的是一个数据宽度为12位、深度为256的ROM，那么可以使用一个256×16的RAM块，忽略高4位数据即可。最终的嵌入式处理器架构如图6-2所示。

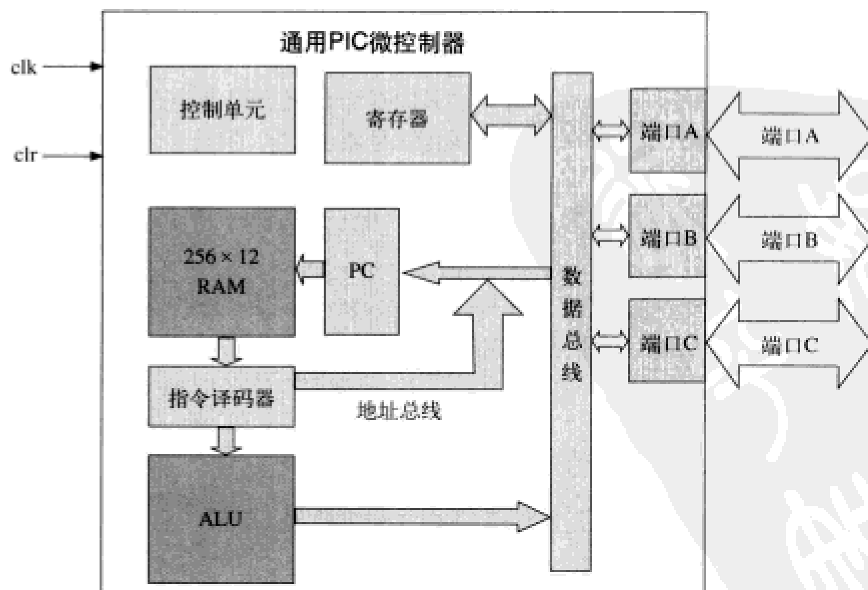



图6-2 嵌入式微控制器架构

## 6.2.2 基本指令

不管微处理器是哪种类型，我们为处理器编写的代码共有3种不同的表示方法，即机器码（二进制数1和0）、汇编代码（低级指令，例如LOAD、STORE等）和高级代码（例如C、FORTRAN或PASCAL）。不管使用什么语言编程，程序代码最终总是被编译或汇编成最低级 59

的机器码并存储在存储器中。高级代码（例如C）被编译为汇编代码，而汇编代码则被汇编成针对某种平台的机器码。

关于编译器的详细解释超出了本书的范围，但是我们可以看一看汇编的基本过程，这对于后面的讨论非常有帮助。

每一种处理器都有一个基本的"指令集"，也就是该处理器可以完成的功能列表。举一个简单的例子，以下是一个伪代码表达式：

$$b=a+2$$

在本例中，变量$a$先加2，然后将结果存储到变量$b$中。在处理器内部，一个变量就是一个可以存储变量值的存储位置，所以我们用下面的汇编语句将变量载入：

LOAD a

这条语句的具体含义是什么呢？无论何时，只要我们从存储器中得到一个变量值，那就意味着将这个变量的值放在了被称为累加器（ACC）的寄存器内。命令LOAD a用自然语言表示就是"将变量$a$表示的存储器位置内的值放在累加器ACC内"。

下一步是将数字2加到累加器中。这是一种简单的方法，因为不使用地址，而是直接将数值加到累加器的当前值上。汇编语言写成的指令可能如下：

ADD #x02

注意，我们使用$x$来表示十六进制数。如果希望加一个变量$c$，那么语句还是相同的，只不过用地址$c$代替了绝对的数值而已。语句如下：

ADD c

现在，累加器内存放的是（$a+2$）的值，这可以存储在某个存储器位置上，或者放在某个端口上（例如端口A）。值得注意的是，数字前使用#表示加的是数值，而不是作为地址使用。

60

在伪代码的例子中，将加法的结果存储到了变量$b$中，命令如下所示：

STORE b

这只是对指令集的一个粗浅的定义，任何处理器都有一个专门的指令集设计细节。还要考虑指令集的数量和数据宽度。如果指令集中指令的数量用$I$表示，那么指令中的操作码（用$n$表示）位数必须满足以下条件：

$$2^n \leqslant I \tag{6-1}$$

换句话说，位数决定了可以定义的不同操作码的数量，也就是定义了指令集可能的大小。例如，如果$n=3$，那么3位二进制数可以表示8种不同的操作码，因此指令集的最大大小为8。

### 6.2.3 取指执行周期

处理器执行程序的标准方法是：将程序存放在存储器内，然后严格按照程序的顺序执行指令。第一步是用程序计数器（PC）递增，以指向要执行的程序，这将以正确的顺序从存储器中调入下一条指令，然后指令被载入到适当的寄存器内去执行。这一步骤称为"取指令执行周期"。

这一阶段会发生什么事儿呢？首先，PC中的内容被载入到存储器地址寄存器（Memory Address Register，MAR）中，然后将存储器内该地址处的数据载入到存储器数据寄存器（Memory Data Register，MDR）。再将MDR寄存器内的数据传送到指令寄存器（Instruction

Register，IR）内。之后，处理器内的PC就可以加1（或者在PC被载入到MDR后立即加1）。

一旦操作码（有时带有参数）被载入，指令就可以立即执行。从本质上来说，每条指令都有自己的状态机和控制路径，并且都连接到了指令寄存器（IR）和时序控制器上（时序控制器定义了与该条指令相关的存储器与寄存器间的数据移动所需要的全部控制信号）。下一节我们将讨论寄存器，除了上面提到的程序计数器（PC）、指令寄存器（IR）和累加器（ACC）外，最少还需要两个存储器寄存器，即MDR和MAR。 61

例如，考虑前面例子中提到的一个简单的命令LOAD a。实际执行这条指令时需要些什么呢？首先，对操作码进行译码，操作码定义了这是一条“载入”命令。接下来确定地址，因为指令中没有使用“#”表示的绝对地址，地址是存储在变量*a*中。下一步，将变量*a*所指向的存储器位置中的数值载入到MDR内，方法是设置MAR = *a*，然后从RAM中得到*a*的值，再将该值传输到累加器（ACC）中。

### 6.2.4 嵌入式处理器的寄存器分配

寄存器的设计，部分取决于我们是希望“克隆”一个PIC器件，还是要创建一个具有更多客户化行为的修改版本。在这两种情况下，有一些寄存器是必须定义的。我们假定需要一个累加器（ACC）、一个程序计数器（PC）和三个输入输出端口（PORTA、PORTB和PORTC）。另外，还要定义指令寄存器（IR）、存储器地址寄存器（MAR）和存储器数据寄存器（MDR）。

除了端口数据外，还需要定义端口的方向，这需要另外三个寄存器来管理三态缓冲器（buffer）输出到数据总线或者从端口输入，三个寄存器分别为DIRA、DIRB和DIRC。此外，还可以定义一些通用寄存器。一般情况下，寄存器的名称、顺序和数量不存在什么问题，但是如果我们想以某种特定器件作为模板，并有可能使用与该器件相同的代码，那么寄存器的配置方式与模板器件完全相同就显得至关重要。

在本例中，我们并没有使用参考器件，因此也不用担心什么，可以定义24个通用寄存器，名称分别为REG0到REG23。与这些通用寄存器相关的是，还需要一个小的译码器来选择正确的寄存器并将其中存储的内容放在数据总线上。

### 6.2.5 一个基本的指令集

为了让器件作为处理器正确地工作，我们必须以指令集的形式定义一些基本的指令。在本例中，我们定义了一些非常基本的指令，这些指令可以执行基本的程序功能，例如ALU功能、存储器功能等。下表是指令集的总结。 62

| 命　令 | 描　述 |
|---|---|
| LOAD arg | 该命令将一个参数载入到累加器内。如果参数有前缀#，那么就是一个绝对数字，否则就是一个相关内存地址。例如：<br>LOAD #01<br>LOAD abc |
| STORE arg | 该命令将累加器中的数据存入存储器内。如果参数有前缀#，那么就是一个绝对地址，否则就是一个相关内存地址。例如：<br>STORE #01<br>STORE abc |

（续）

| 命　令 | 描　述 |
|---|---|
| ADD arg | 该命令将参数加到累加器中。如果参数有前缀#，那么就是一个绝对数字，否则就是一个相关内存地址。例如：<br>ADD #01<br>ADD abc |
| NOT | 该命令对累加器执行"非"操作。 |
| AND arg | 该命令将参数和累加器进行"与"操作。如果参数有前缀#，那么就是一个绝对数字，否则就是一个相关内存地址。例如：<br>AND #01<br>AND abc |
| OR arg | 该命令将参数和累加器进行"或"操作。如果参数有前缀#，那么就是一个绝对数字，否则就是一个相关内存地址。例如：<br>OR #01<br>OR abc |
| XOR arg | 这一命令将参数和累加器进行"异或"操作。如果参数有前缀#，那么就是一个绝对数字，否则就是一个相关内存地址。例如：<br>XOR #01<br>XOR abc |
| INC | 这一命令对累加器进行加 1 操作 |
| SUB arg | 这一命令执行从累加器减去参数的操作。如果参数有前缀#，那么就是一个绝对数字，否则就是一个相关内存地址。例如：<br>SUB #01<br>SUB abc |
| BRANCH arg | 这一命令可以令程序跳转到程序中的某个特定的位置执行。这对于循环等程序结构非常有用。如果参数有前缀#，那么就是一个绝对地址，否则就是一个相关内存地址。例如：<br>BRANCH #01<br>BRANCH abc |

在这个简单的指令集中，有10条不同的指令。这就是说，按照前面公式（6-1）计算，至少需要4bit来表示上表中给出的各条指令。假如我们要用8bit表示数据，那么存储程序的ROM至少必须有12bit的位宽。为了扩充更多的指令，同时也为了处理不同地址模式之间的差异（例如，绝对数与变量的不同），我们建议使用16bit的存储系统。

63～64

注意，在这一阶段还没有定义端口和寄存器。稍后我们将把模型进行扩充以处理这些行为。

## 6.2.6 结构级还是行为级

目前为止，在简单微处理器的设计中，我们对细节的讨论还仅局限在用寄存器和总线这样的术语来相当抽象地描述处理器。在这一阶段，我们只是决定了程序和架构在设计方面如何实现的问题。

一种选择是，将汇编语言写成的程序转换为一个可以用VHDL很容易实现的状态机，以测试算法。使用这种方法，以简单的规则限定代码只使用适应于特定处理器的寄存器和技术，

就可以很容易地修改程序并重新编译。这对于研究和开发算法可能非常有用，但是相比最终的设计实现可能更显得理想化一些，因为实际的设计中有存储器访问造成的延迟，还有存储器的配置以及一些控制信号等，所以这时使用专用硬件设计更合适。

另一种选择是，开发一个简单的处理器模型，具有处理器最终实现的一部分特征，不过仍然使用汇编语言描述的模型来测试。这样有一些优点，因为不需要编译成机器码。但是这还不能反应最终的处理器架构的一些详细硬件特征，从而可能引起实际应用中的一些问题。

第三种选择是，开发结构级的处理器模型，机器码可以直接从ROM中读取。这是一种非常好的方法，对于检查程序非常有用，同时也有助于发现软硬件结合起来后可能存在的问题，因为这种模型的架构直接反映了将要在FPGA中实现的模型结构。

### 6.2.7 机器码指令集

为了给我们的处理器创建一套合适的指令集，汇编语言指令集还需要有等价的机器码指令集，后者可以在处理器内部由序列器进行译码。最终得到的操作码/指令表如下所示。

65

| 指　　令 | 操作码（二进制） |
|---|---|
| LOAD arg | 0000 |
| STORE arg | 0001 |
| ADD arg | 0010 |
| NOT | 0011 |
| AND arg | 0100 |
| OR arg | 0101 |
| XOR arg | 0110 |
| INC | 0111 |
| SUB arg | 1000 |
| BRANCH arg | 1001 |

### 6.2.8 微处理器的结构单元

以图6-2中给出的微处理器抽象设计为基础，用图6-3所示结构中的精确寄存器和总线配

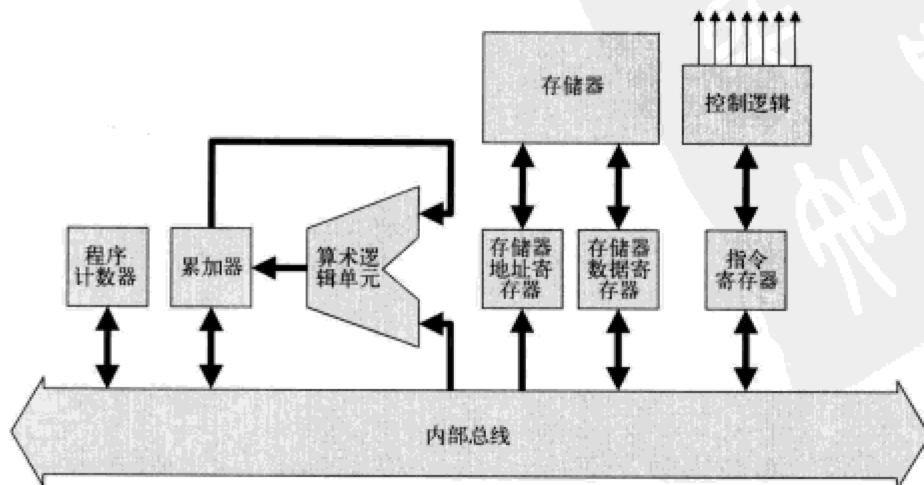



图6-3 微处理器的结构化模型

66 置进行重画。然后就可以使用这一模型为连接到内部总线上的每个模块创建各自的VHDL模型，再设计一个控制模块，依次去处理所有相关的序列和每个模块的控制标志。

然而，在开始之前，一定要定义模型的基本准则才有意义。首要的是定义基本数据类型。在任何数字模型中（如本书其他章节所示），明智的做法是保证数据可以在标准模型之间传递，因此，在这个例子中我们将使用std_logic_1164库，因为它是数字模型的标准。

为了使用这个库，每一个信号需要定义成基本类型std_logic，而且在处理器每个模型的开头部分要声明库ieee.std_logic_1164.all。

最后，处理器中每个模型要定义成单独的模型，方便VHDL实现。

### 6.2.9 处理器函数包

为了简化每个独立模块的VHDL设计，定义了一组标准函数，并组成一个包，称为处理器函数，主要用来为这组模型定义有用的类型和函数。VHDL包如下所示：

```
Library ieee;
Use ieee.std_logic_1164.all;

Package processor_functions is
      Type opcode is ( load, store, add, not, and, or, xor, inc, sub, branch );
      Function Decode ( word : std_logic_vector ) return opcode;
      Constant n : integer : = 16;
      Constant oplen : integer : = 4;
      Type memory_array is array (0 to 2** ( n-oplen-1 ) of Std_logic_vector(n-1
        downto 0);
      Constant reg_zero : unsigned ( n-1 downto 0 ) : = ( others => '0' );
End package processor_functions;

Package body processor_functions is
      Function Decode ( word : std_logic_vector ) return
        opcode is
            Variable opcode_out : opcode;
      Begin
            Case word ( n-1 downto n-oplen-1 ) is
                When "0000" => opcode_out : = load;
                When "0001" => opcode_out : = store;
                When "0010" => opcode_out : = add;
                When "0011" => opcode_out : = not;
                When "0100" => opcode_out : = and;
                When "0101" => opcode_out : = or;
                When "0110" => opcode_out : = xor;
                When "0111" => opcode_out : = inc;
                When "1000" => opcode_out : = sub;
                When "1001" => opcode_out : = branch;
                When others => null;
            End case;
```

67

```
            Return opcode_out;
        End function decode;
    End package body processor_functions;
```

### 6.2.10 程序计数器

程序计数器（PC）必须有一个系统时钟、复位和系统总线（定义为inout类型，以便PC寄存器模块读写）。另外，还需要一些控制信号。第一个信号是PC递增信号PC_inc，第二个信号是载入特定值到PC中的控制信号PC_load，最后一个信号是指示寄存器内容在内部总线上有效的信号PC_valid。通过这一信号可以保证，当处理器总线上不需要寄存器的值时，PC寄存器的值将是高阻（Z）。系统总线（PC_bus）定义为双向的std_logic_vector类型，这样既可以读也可以写。最终得到的VHDL实体如下所示：

```
library ieee;
use ieee.std_logic_1164.all;
entity pc is
      Port (
            Clk : IN std_logic;
            Nrst : IN std_logic;
            PC_inc : IN std_logic;
            PC_load : IN std_logic;
            PC_valid : IN std_logic;
            PC_bus : INOUT std_logic_vector(n-1 downto 0)
      );
End entity PC;
```

程序计数器必须能够处理控制信号的各种配置情况，以及与内部总线进行正确的数据通信。程序计数器模型既有异步逻辑，又有同步逻辑。如果信号PC_valid变为低电平，那么PC_bus信号的所有位应该被设置为高阻Z。当然，如果复位信号变低，程序计数器应该复位到0。 [68]

程序计数器的同步逻辑是递增和载入功能。当时钟上升沿到来时，信号PC_load和PC_inc控制计数器的功能。递增功能具有优先级，如果PC_inc为高，忽略载入（load）功能，计数器将增加。如果不启用递增功能（即PC_inc为低），同时PC_load为高，那么PC将把总线上的当前值载入。

VHDL描述如下：

```
architecture RTL of PC is
      signal counter : unsigned (n-1 downto 0);
begin
      PC_bus <= std_logic_vector(counter)
            .     when PC_valid = '1' else (others => 'Z');
      process (clk, nrst) is
      begin
            if nrst = '0' then
                    count <= 0;
            elsif rising_edge(clk) then
                  if PC_inc = '1' then
```

```
                    count <= count + 1;
                else
                    if PC_load = '1' then
                        count <= unsigned(PC_bus);
                    end if;
                end if;
            end if;
        end process;
    end architecture RTL;
```

### 6.2.11 指令寄存器

指令寄存器（IR）具有与PC相同的时钟和复位信号，与总线的接口（IR_bus）同样定义为std_logic_vector类型的双向接口（INOUT）。指令寄存器也有两个控制信号，第一个是载入指令（IR_load），第二个是将地址放在系统总线上（IR_address）。最后连接的是将要发送给系统控制器的译码后的操作码。指令寄存器定义为与系统总线相同宽度的简单无符号整数类型。指令寄存器的VHDL实体描述如下所示：

69

```
library ieee;
use ieee.std_logic_1164.all;
use work.processor_functions.all;
entity ir is
    Port (
        Clk : IN std_logic;
        Nrst : IN std_logic;
        IR_load : IN std_logic;
        IR_valid : IN std_logic;
        IR_address : IN std_logic;
        IR_opcode : OUT opcode;
        IR_bus : INOUT std_logic_vector(n-1 downto 0)
    );
End entity IR;
```

指令寄存器的功能是对二进制形式的操作码进行译码，然后发送给控制模块。如果IR_valid为低，总线应该被设置为高阻。如果复位信号（nrst）为低，那么寄存器复位为全0。

在时钟的上升沿，总线上的数据将被发送到内部寄存器中，当指令寄存器内的数据发生变化时，对操作码进行异步译码。

VHDL构造体如下：

```
architecture RTL of IR is
    signal IR_internal : std_logic_vector (n-1 downto 0);
begin
    IR_bus <= IR_internal
        when IR_valid = '1' else (others => 'Z');
    IR_opcode <= Decode(IR_internal);
    process (clk, nrst) is
    begin
```

```
            if nrst = '0' then
                IR_internal <= (others => '0');
            elsif rising_edge(clk) then
                if IR_load = '1' then
                    IR_internal <= IR_bus;
                end if;
            end if;
        end process;
    end architecture RTL;
```

在以上VHDL描述中，我们使用了预定义的函数Decode，这一函数在前面定义的处理器功能包（processor_functions package）中定义。这一函数对送给指令寄存器的高4位数据（即操作码）进行译码，然后发送给控制器。

70

## 6.2.12 算术和逻辑单元

算术和逻辑单元（ALU）也具有与程序计数器相同的时钟和复位信号，与总线的接口（ALU_bus）定义为std_logic_vector类型的双向接口（INOUT）。算术和逻辑单元有3个控制信号，译码后可以映射为8个独立的ALU功能。算术和逻辑单元还包含一个累加器（ACC），数据宽度与系统总线相同，类型为std_logic_vector。另外，还有一个单位输出ALU_zero，当累加器中的数据为0时，ALU_zero信号变为高电平。

算术和逻辑单元的VHDL描述如下：

```
library ieee;
use ieee.std_logic_1164.all;
use work.processor_functions.all;
entity alu is
        Port (
             Clk  : IN std_logic;
             Nrst  : IN std_logic;
             ALU_cmd  : IN std_logic_vector(2 downto 0 );
             ALU_zero  : OUT std_logic;
             ALU_valid  : IN std_logic;
             ALU_bus  : INOUT std_logic_vector(n-1 downto 0 )
        );
End entity alu;
```

算术和逻辑单元的功能是对二进制形式的ALU_cmd输入进行译码，然后根据译码结果对总线上的数据和累加器内的数据执行相关操作。如果信号ALU_valid为低，总线将是高阻态。如果复位信号（nsrt）为低，那么内部寄存器的值都变为0。

在时钟上升沿，总线数据将被发送到内部寄存器中，并对指令进行译码。

相关的VHDL构造体代码如下：

```
architecture RTL of ALU is
      signal ACC : std_logic_vector (n-1 downto 0);
begin
      ALU_bus <= ACC
            when ACC_valid = '1' else (others => 'Z');
```

71

```
    ALU_zero <= '1' when acc = reg_zero else '0';
    process (clk, nrst) is
    begin
        if nrst = '0' then
            ACC <= (others => '0');
        elsif rising_edge(clk) then
            case ACC_cmd is
            -- Load the Bus value into the accumulator
            when "000" => ACC <= ALU_bus;
            -- Add the ACC to the Bus value
            When "001" => ACC <= add(ACC,ALU_bus);
            -- NOT the Bus value
            When "010" => ACC <= NOT ALU_bus;
            -- OR the ACC to the Bus value
            When "011" => ACC <= ACC or ALU_bus;
            -- AND the ACC to the Bus value
            When "100" => ACC <= ACC and ALU_bus;
            -- XOR the ACC to the Bus value
            When "101" => ACC <= ACC xor ALU_bus;
            -- Increment ACC
            When "110" => ACC <= ACC + 1;
            -- Store the ACC value
            When "111" => ALU_bus <= ACC;
        end if;
    end process;
end architecture RTL;
```

### 6.2.13 存储器

处理器需要一个随机存取存储器（RAM）、一个地址寄存器（MAR）和一个数据寄存器（MDR）。因此，每个寄存器都需要一个载入（load）信号：MDR_load和MAR_load。由于有存储器，还要有一个使能信号（M_en）和一个读写信号（M_rw）。最后，与系统总线的连接是一个标准的双向端口（同微处理器中为其他寄存器定义的一样）。

存储器模块的VHDL实体描述如下：

```
library ieee;
use ieee.std_logic_1164.all;
use work.processor_functions.all;
entity memory is
    Port (
        Clk : IN std_logic;
        Nrst : IN std_logic;
        MDR_load : IN std_logic;
        MAR_load : IN std_logic;
        MAR_valid : IN std_logic;
        M_en : IN  std_logic;
```

```
            M_rw : IN std_logic;
            MEM_bus : INOUT std_logic_vector(n-1downto 0)
        );
End entity memory;
```



存储器模块有3方面的功能。第一个是存储器地址载入MAR寄存器。第二个是使用MDR寄存器进行存储器读写操作。最后一个是存储处理器将要运行的程序。在VHDL模型中，我们将使用一个常数数组实现存储器，并存储程序数据。

存储器的VHDL构造体如下所示：

```
architecture RTL of memory is
    signal mdr : std_logic_vector(wordlen-1 downto 0);
    signal mar : unsigned(wordlen-oplen-1 downto 0);
              begin
    MEM_bus <= mdr
        when MEM_valid = '1' else (others => 'Z');
    process (clk, nrst) is
        variable contents : memory_array;
        constant program : contents :=
        (
            0  =>  "0000000000000011",
            1  =>  "0010000000000100",
            2  =>  "0001000000000101",
            3  =>  "0000000000001100",
            4  =>  "0000000000000011",
            5  =>  "0000000000000000" ,
            Others => (others => '0')
        );
begin
        if nrst = '0' then
            mdr <= (others => '0');
            mdr <= (others => '0');
            contents := program;
        elsif rising_edge(clk) then
            if MAR_load = '1' then
                mar <= unsigned(MEM_bus(n-oplen-1 downto 0));
            elsif MDR_load = '1' then
                mdr <= MEM_bus;
            elsif MEM_en = '1' then
                if MEM_rw = '0' then
                    mdr <= contents(to_integer(mar));
                else
                    mem(to_integer(mar)):= mdr;
                end if;
```

```
                end if;
            end if;
        end process;
    end architecture RTL;
```

73

我们可以找一些更详细的VHDL资料来说明这一阶段会完成哪些工作。存储器模块中有两个内部信号：mdr和mar（分别是数据和地址）。第一个需要注意的地方是：前面我们已经将地址寄存器MAR定义为无符号数，而不是std_logic_vector类型，但是可以直接进行索引。数据寄存器MDR是std_logic_vector类型。可以直接使用整数类型的数据，它们可以很容易地转换为std_logic_vector类型。

```
signal mdr : std_logic_vector(wordlen-1 downto 0);
signal mar : unsigned(wordlen-oplen-1 downto 0);
```

第二个需要注意的地方是程序本身。很明显，程序有可能很大，但是在当前的例子中，我们只定义了一个简单的只有3个变量的程序：

$$c = a + b$$

二进制代码如下：

```
0 => "0000000000000011",
1 => "0010000000000100",
2 => "0001000000000101",
3 => "0000000000001100",
4 => "0000000000000011",
5 => "0000000000000000" ,
Others => (others => '0')
```

例如，考虑地址0处的一行代码，16位数据为0000000000000011。我们可以将其分解为操作码和数据两部分：

操作码　　0000

数据　　000000000011 (3)

换句话说，这条指令的意义是从地址3处载入数据。类似地，第二行指令为加地址4处的数据，第三行指令是将结果存储到地址5处。地址3、4和5处分别存储3个变量。

## 6.2.14 微控制器

74

处理器的细节操作由一个序列控制器（控制模块）控制。处理器这一部分的功能是得到当前PC地址，从存储器中查找相关的指令，得到需要的数据，在正确的时间、用正确的值设置所有相关的控制信号。

因此，控制器必须有一个时钟和复位信号（与设计中的其他模块相同），还要与系统总线连接，所有的控制信号必须输出。控制器的例子如下：

```
library ieee;
use ieee.std_logic_1164.all;
use work.processor_functions.all;
entity controller is
    generic (
        n : integer := 16
```

```
    );
    Port (
        Clk : IN std_logic;
        Nrst : IN std_logic;
        IR_load : OUT std_logic;
        IR_valid : OUT std_logic;
        IR_address : OUT std_logic;
        PC_inc : OUT std_logic;
        PC_load : OUT std_logic;
        PC_valid : OUT std_logic;
        MDR_load : OUT std_logic;
        MAR_load : OUT std_logic;
        MAR_valid : OUT std_logic;
        M_en : OUT std_logic;
        M_rw : OUT std_logic;
        ALU_cmd : OUT std_logic_vector(2 downto 0);
        CONTROL_bus : INOUT std_logic_vector(n-1 downto 0)
    );
End entity controller;
```

通过这个实体，每个模块所需要的控制信号都有了定义，使用这些控制信号可以执行程序的各项功能。控制器的构造体（architecture）使用一个基本的状态机来实现，以驱动各个信号。处理器的状态机定义如图6-4所示。



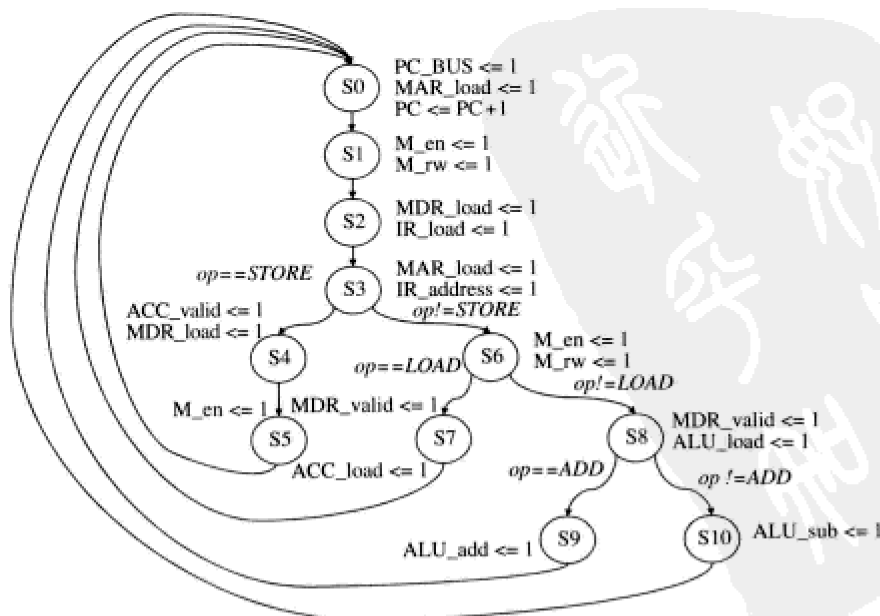



图6-4 处理器的控制器状态机

我们可以用一个基本的VHDL构造体来实现这一功能，用一种新的状态类型实现每个状态，用case语句管理状态机的转换流程。VHDL构造体如图6-4所示，包括状态机的同步控制

部分（时钟与复位）和下一状态的产生逻辑。

```
architecture RTL of controller is
      type states is
         (s0,s1,s2,s3,s4,s5,s6,s7,s8,s9,s10);
      signal current_state, next_state : states;
begin
         state_sequence: process (clk, nrst) is
              if nrst = '0' then
                    current_state <= s0;
              else
                   if rising_edge(clk) then
                       current_state <=
                          next_state;
                   end if;
              end if;
         end process state_sequence;

         state_machine : process ( present_state,
           opcode ) is
              -- state machine goes here
         End process state_machine;
end architecture;
```

76

从这段VHDL代码可以看出，第一个process（state_sequence）处理当前状态到下一状态的转换和复位过程。注意，这是一个同步状态机，所有转换都在时钟的上升沿发生，但复位是异步的。第二个process（state_machine）等待状态的变化或者操作码的变化，用来管理下一状态的产生，不过状态转换本身实际上是在第一个process中完成的。相应的VHDL代码如下：

```
state_machine : process ( present_state,
  opcode ) is
begin
       -- Reset all the control signals
       IR_load <= '0';
       IR_valid <= '0';
       IR_address <= '0';
       PC_inc <= '0';
       PC_load <= '0';
       PC_valid <= '0';
       MDR_load <= '0';
       MAR_load <= '0';
       MAR_valid <= '0';
       M_en <= '0';
       M_rw <= '0';
       Case current_state is
       When s0 =>
```

```
          PC_valid <= '1'; MAR_load <= '1';
          PC_inc <= '1'; PC_load <= '1';
          Next_state <= s1;
       When s1 =>
          M_en <= '1'; M_rw <= '1';
          Next_state <= s2;
       When s2 =>
          MDR_valid <= '1'; IR_load <= '1';
          Next_state <= s3;
       When s3 =>
          MAR_load <= '1'; IR_address <= '1';
          If opcode = STORE then
                Next_state <= s4;
          else
                Next_state <=s6;
          End if;
       When s4 =>
          MDR_load <= '1'; ACC_valid <= '1';
          Next_state <= s5;
       When s5 =>
          M_en <= '1';
          Next_state <= s0;
       When s6 =>
          M_en <= '1'; M_rw <= '1';
          If opcode = LOAD then
                 Next_state <= s7;
          else
                 Next_state <= s8;
          End if;
          When s7 =>
             MDR_valid <= '1'; ACC_load <= '1';
             Next_state <= s0;
          When s8 =>
             M_en<= '1'; M_rw <= '1';
             If opcode = ADD then
                    Next_state <= s9;
             else
                    Next_state <= s10;
             End if;
          When s9 =>
             ALU_add <= '1';
             Next_state <= s0;
          When s10 =>
             ALU_sub <= '1';
             Next_state <= s0;
       End case;
End process state_machine;
```

### 6.2.15 简单微处理器总结

到目前为止，处理器中一些重要的部件都已经定义完成了。在基本的VHDL网表中例化它们并用这些模块构造一个处理器现在已经是一件非常简单的事情。通过改变地址/数据总线宽度，或者扩展指令集来修改处理器的功能也是很简单的事情。

## 6.3 FPGA中的软核处理器

前面给出的简单微处理器设计实例对于设计练习很有用，同时也有助于理解微处理器的工作原理和过程。实际上，大多数FPGA厂商都将一些标准的处理器内核作为嵌入式开发套件（包括编译器和一些库）的一部分提供给用户。例如，Xilinx公司的Microblaze处理器内核以及Altera公司的NIOS处理器内核等。在上述情况中，基本的概念是相同的，都是在设计中实例化一个标准的可配置内核，然后使用标准的编译器编译代码，最后下载到FPGA中。

78

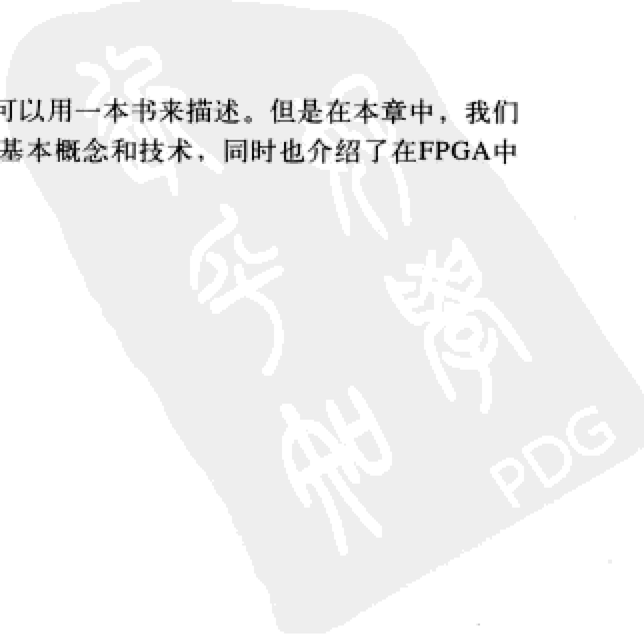
每个处理器软核都是不同的，不同应用中具体的细节也不同，在这一节中，我们只讨论一般原理，建议读者用FPGA厂商提供的开发板进行实验，看一看哪种更适合自己的应用程序。

任何软核开发系统中都有几个关键的功能，这些功能使得开发流程更加容易实现。第一个是构建系统的功能。通过这一功能，可以将一个软核集成到一个完整的硬件系统中，该系统包括存储器、控制模块、DMA模块、数据接口和中断。第二个是处理器类型的选择。一个基本的NIOS II内核或者其他类似的嵌入式内核，其典型的性能在100~200MIPS之间，处理器设计工具可以根据可用的硬件资源以及性能需求对内核的大小进行适当调整。

## 6.4 小结

有关FPGA中的嵌入式处理器的内容，完全可以用一本书来描述。但是在本章中，我们只讨论了在FPGA中直接实现一个简单处理器的基本概念和技术，同时也介绍了在FPGA中实现软核的方法。

79



# 第三部分　设计工具箱

本书第三部分介绍设计工具箱，其中都是些非常标准的功能，常用于各种设计和测试电路中，因此，至少对初始设计进行评估时极其有用，而不用完全从头开发。

第7章是这一部分的第一个章节，主要介绍串行通信，从数据传输的基础知识入手，继而讨论了如何将USB集成到设计中的实际方法。第8章讨论了数字滤波器，用一个简单的例子说明了如何处理标准的拉普拉斯（S域）描述，并且用VHDL数字滤波器实现它。第9章介绍了一个越来越重要的主题——安全系统，具体说明了块加/解密器DES和AES。后半部分还涉及了一些标准的接口。第10章主要描述了如何用VHDL进行存储器建模，同时也描述了ROM、RAM、Flash和SRAM的一些内容。第11章和第12章分别描述了如何实现一个简单的PS/2鼠标接口和键盘接口，回顾了一些数据模式和协议，并描述了它们的简单实现问题。最后，第13章说明了如何构建一个简单的VGA接口，并用VHDL完成了同步设计代码。

可能有许多读者需要开发比较复杂的应用，但是往往不得不从头开始构建系统框架，而他们又不希望从头开发所有的常规功能，那么这一章对于他们来说无疑是一个有用的起点。本部分可以帮助读者加快学习进度，但是也要强调，书中所给的例子纯粹是为了教学目的，并且作者在写本书时也尽可能做了简化，以求清晰明了。

# 第7章 串行通信

## 7.1 引言

目前存在各种各样的串行通信协议，但是所有的协议都依赖于某种形式的编码方案，以便通过传输媒介高效而准确地传输串行数据。本章首先回顾了常用的数据传输方法，如RS-232和通用串行接口（USB），然后讨论了一些有用的编码方法，例如曼彻斯特（Manchester）编码、符号反转（Code Mark Inversion，GMI）编码、不归零（Non-Return-toZero，NRZ）编码、不归零反转（Non-Return-toZero-Inverted，NRZI）编码等，它们常常用作高级传输协议的一部分。例如，NRZI编码用在USB协议中。

## 7.2 曼彻斯特编解码

曼彻斯特编码是一种简单的编码方案，它将一个基本比特流翻译成为一系列的转换。这种编码对于保证数据传输具有一定的带宽非常有用，因为不管数据序列是什么，传输流的频率总是原始数据流的两倍。另外，信号恢复也非常简单，因为完全没有必要提取时钟信号，只要找到信号的边缘进行异步提取就可以实现数据的恢复。曼彻斯特编码的基本方法如图7-1所示。

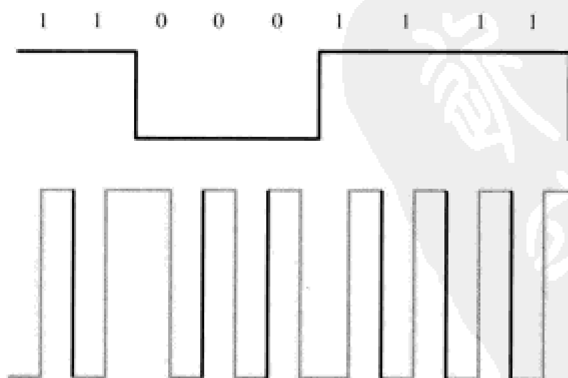



图7-1 曼彻斯特编码方案

这种编码方案的另一个优点是具有非常好的数据容错性能。如果在传输过程中发生错误，后面的数据根本不会受到任何影响，不管这个错误发生在发送端、传输媒介中还是接收端，短暂的干扰脉冲之后，数据可以继续传输，而不用进行错误纠正。当然，原始数据可以采用某种形式的编码以增强其纠错能力（见本书关于数据检查方法的章节，例如奇偶校验和CRC等）。

如果我们要为这种类型的编码创建一个VHDL模型，实际上是非常简单的。第一步是确认要有单比特数据输入D和时钟信号CLK。为什么要用同步电路？因为使用同步时钟，我们

才可以在时钟的上升沿定义对输入数据的采样点，才可以在时钟的上升沿和下降沿定义输出的转换。行为级VHDL代码如下所示：

```
Library ieee;
Use ieee.std_logic_1664.all;

Entity Manchester_encoder is
      Port (
           Clk : in std_logic;
           D : in std_logic;
           Q : out std_logic
      );
End entity Manchester_encoder;

Architecture basic of Manchester_encoder is
      Signal lastd : std_logic := '0';
Begin
      P1: Process ( clk )
      Begin
           If rising_edge(clk) then
                if ( d = '0' ) then
                       Q <= '1';
                       Lastd <= '0';
                elsif ( d = '1' ) then
                       Q <= '0';
                       Lastd <= '1';
           Else
                       Q <= 'X';
                       Lastd <= 'X';
                      End if;
           End if;
      If falling_edge(clk) then
           If ( lastd = '0' ) then
                       Q <= '0';
           elsif ( lastd = '1' ) then
                       Q <= '1';
      Else
                       Q <= 'X';
           End if
      End if;
   End process p1;
End architecture basic;
```



可见，VHDL代码是非常简单的，但是还有一个更加简单的方法可以实现对数据的编码，那就是将时钟信号和数据进行异或操作。还以同样的数据序列为例，我们将看到，如果增加一个时钟信号并观察原始数据和曼彻斯特编码输出，就会发现这其实是一种数据与时钟的异

或关系，如图7-2所示。

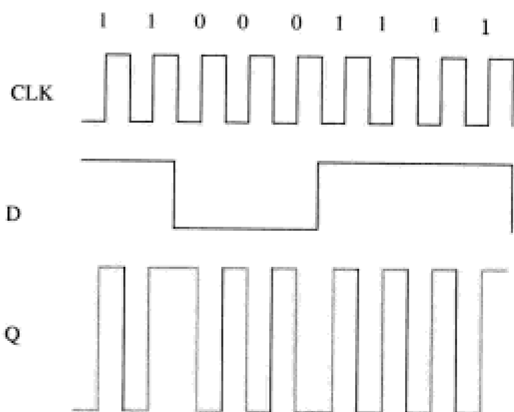



图7-2 使用异或功能实现的曼彻斯特编码

因此，用这种简单的方案可以创建一个更加简单的曼彻斯特编码器，只要将时钟和数据进行异或即可得到曼彻斯特编码数据流，相应的VHDL代码如下：

```
Library ieee;
Use ieee.std_logic_1664.all;

Entity Manchester_encoder is
      Port (
            Clk : in std_logic;
            D : in std_logic;
            Q : out std_logic
      );
End entity Manchester_encoder;

Architecture basic of Manchester_encoder is
Begin
      Q <= D XOR CLK;
End architecture basic;
```

对曼彻斯特数据流的译码也有同步和异步两种方法。我们可以使用一个本地时钟来探测输入数据的值，并分别估计上升沿和下降沿的值是0还是1，然后确定输入数据，但是这显然依赖于发送器和接收器的时钟被同步到了一个合理的程度。这样一个简单的译码器如下所示：

85

```
Entity Manchester_decoder is
      Port (
            Clk : in std_logic;
            D : in std_logic;
            Q : out std_logic
      );
End entity Manchester_decoder;
```

```
Architecture basic of Manchester_decoder is
      Signal lastd : std_logic := '0';
Begin
      P1 : process (clk)
      Begin
           If rising_edge(clk) then
                 Lastd <= d;
           End if;
      If falling_edge(clk) then
           If (lastd = '0') and (d = '1') then
                 Q <= '1';
           Elsif (lastd = '1') and (d = '0') then
                 Q <= '0';
           Else
                 Q <= 'X';
           End if;
      End if;
    End process p1;
End architecture basic;
```

在VHDL模型中，时钟与发送器是相同的频率，并且数据应该以包（packet）的形式发送，以保证发送的数据块不要太大，否则时钟可能会失去同步，而且数据要能够进行完整性检查，以便纠正错误或者接收端时钟出现相位偏移。

86

## 7.3 不归零编解码

不归零编码实际上根本不能算作一种编码方案，因为这种编码直接按原码发送，即发送0就是数据0，发送1就是数据1。之所以提到这种编码，只是因为设计人员可能看到NRZ这个术语，会假定需要使用某种编解码器，但事实上并非如此。另外还要注意一点，这种编码方法虽然简单，却存在一些重大的缺陷。第一个缺陷（尤其是与曼彻斯特编码相比）是，发送长0和长1序列时会产生直流电平，这非常容易被噪声影响，而且从信号中提取时钟也非常困难。另一个问题是带宽。如果与曼彻斯特编码相比，很明显就可以看出，曼彻斯特编码需要相当窄的带宽就可以发送数据，而NRZ编码可能需要从直流电平到半数据率（奈奎斯特频率）及其之间的所有频率。这就使得线路设计和滤波器设计非常困难。

## 7.4 不归零反转编解码

不归零反转码中，不归零编码存在的潜在问题，特别是长时间的直流电平问题只是部分地被解决了。对于不归零反转码，如果数据是0，那么输出数据就不会改变，但是如果数据是1，那么输出将发生改变。因此，长时间的1序列不会产生问题，但是长时间的0仍然存在着原来的问题。使用VHDL编写一个基本的不归零反转编码器模型非常简单，代码如下：

```
Entity nrzi_encoder is
      Port (
            CLK : in std_logic;
```

```
            D : in std_logic;
            Q : out std_logic
        );
End entity nrzi_encoder;

Architecture basic of nrzi_encoder is
        Signal qint : std_logic := '0';
Begin
        p1 : process (clk)
        Begin
            if (d = '1') then
                if ( qint = '0' ) then
                    Qint <= '1';
                else
                    Qint <= '0';
                End if;
            End if;
    End process p1;
    Q <= qint;
End architecture basic;
```

87

注意，以上模型是同步的，如果要改为异步模型，只要将设计中的时钟端口clk去掉，并将过程敏感表中的clk改为d即可。我们可以使用同样的逻辑产生输出，这样就得到了译码数据流，相应的VHDL代码如下，这里我们使用同步方式：

```
Entity nrzi_decoder is
        Port (
            CLK : in std_logic;
            D : in std_logic;
            Q : out std_logic
        );
End entity nrzi_decoder;

Architecture basic of nrzi_decoder is
      Signal lastd : std_logic := '0';
Begin
        p1 : process (clk)
        Begin
            If rising_edge(clk) then
                If (d = lastd) then
                    Q <= '0';
                Else
                    Q <= '1';
                End if;
                Lastd <= d;
            End if;
    End process p1;
End architecture basic;
```

不归零反转解码器也非常简单，因为我们唯一要检查的是，在最后一个时钟沿后，数据流是否发生变化。如果最后一个时钟后数据有变化，那么数据就是1，如果数据没有变化，就是0。显然，也可以使用异步方式实现，但是这严重依赖于下游用于同步的数据检查算法的正确性。

88

## 7.5 RS-232

### 7.5.1 引言

RS-232串行传输的基本方式是UART（Universal Asynchronous Receiver/Transmitter，通用异步收发器）。这是计算机上采用的一种将串行通信数据流转换成并行格式的标准方法。RS-232是UART的一个专门的标准，它定义了开始（start）、停止（stop）、中断（break）、数据（data）、奇偶校验（parity）和管脚名称等。

### 7.5.2 RS-232波特率产生器

RS-232是一个异步传输方案，因此在传输以前必须定义正确的时钟速率，以便能够正确无误地接收或发送数据。RS-232的波特率范围为1 200~115 200波特。这都是以一个频率为14.745 6MHz的标准时钟为基础的，将这个时钟按如下分频比分频即可得到正确的波特率：8，16，28，48，96，192，384和768。我们需要定义一个时钟分频电路，可以通过一个控制字配置输出正确的波特率。总共有8种不同的波特率，因此可以使用一个3比特的控制字（baud[2:0]），再加上一个时钟和复位信号，就可以得到正确的频率。假定基本的时钟频率为14.745 6MHz，见图7-3。

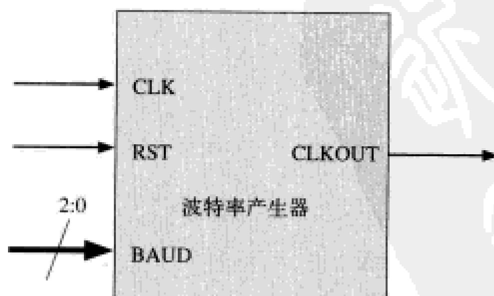



图7-3 波特率时钟产生器

这一控制器的VHDL模型如下所示，其中使用一个过程选择正确的波特率，另一个过程实现对输入时钟的分频：

89

```
LIBRARY ieee;
USE ieee.Std_logic_1164.ALL;
USE ieee.Std_logic_unsigned.ALL;
ENTITY baudcontroller IS
      PORT(
            clk  : IN std_logic;
            rst  : IN std_logic;
            baud : IN std_logic_vector(0 to 2);
```

```
            clkout : OUT std_logic);
END baudcontroller;

ARCHITECTURE simple OF baudcontroller IS
      SIGNAL clkdiv : integer := 0;
            SIGNAL count : integer := 0;
BEGIN
Div: process (rst, clk)
begin
      if rst = '0' then
            clkdiv <= 0;
            count <= 0;
      elsif rising_edge(CLK) then
      case Baud is
            when "000" => clkdiv <= 7; -- 115200
            when "001" => clkdiv <= 15; -- 57600
            when "010" => clkdiv <= 23; -- 38400
            when "011" => clkdiv <= 47; -- 19200
            when "100" => clkdiv <= 95; -- 9600
            when "101" => clkdiv <= 191; -- 4800
            when "110" => clkdiv <= 383; -- 2400
            when "111" => clkdiv <= 767; -- 1200
            when others => clkdiv <= 7;
      end case;
      end if;
end process;

clockdivision : process (clk, rst)
begin
      if rst= '0' then
            clkdiv <= 0;
            count <= 0;
      elsif rising_edge(CLK) then
            count <= count + 1;
            if (count > clkdiv) then
                  clkout <= not clkout;
                  count <= 0;
            end if;
      end if;
end process;
END simple;
```

### 7.5.3　RS-232接收器

[90]　RS-232接收器等待数据到达RX线，发送的数据必须满足如下的规范定义：<若干比特的数据><奇偶校验><停止位>。所以，如果数据为8比特，没有奇偶校验位，1比特停止位，那

么规范将称为8N1。RS-232标准规定的电压范围为−12~+12V，这里假定已经有接口芯片将该电压转换为标准逻辑电压（例如0~5V或者0~3.3V）。位流的格式如图7-4所示。

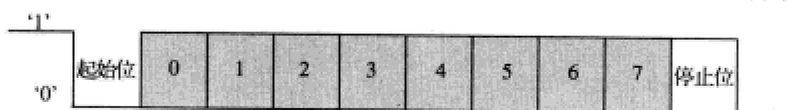



图7-4 串行数据接收器

RS-232的空闲状态是高电平，但图7-4中在停止位之后变为低电平，实际上只有一种可能，那就是另一个数据已经被发送过来了。如果数据传送已经完成，那么数据线将再次变为高电平（即空闲态）。事实上，我们可以用一个简单的状态机来建立一个模型，如图7-5所示。

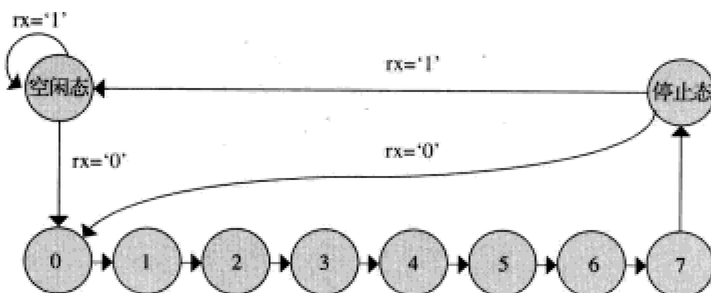



图7-5 基本串行接收器

使用VHDL实现这个简单的状态机，代码如下：

```
LIBRARY ieee;
USE ieee.Std_logic_1164.ALL;
USE ieee.Std_logic_unsigned.ALL;

ENTITY serialrx IS
      PORT(
          clk : IN std_logic;
          rst : IN std_logic;
          rx : IN std_logic;
          dout : OUT std_logic_vector (7 downto 0)
          );
END serialrx;
ARCHITECTURE simple OF serialrx IS
  type state is (idle, s0, s1, s2, s3, s4, s5, s6, s7,
    stop);
  signal current_state, next_state : state;
  signal databuffer : std_logic_vector (7 downto 0);
BEGIN
receive: process (rst, clk)
begin
      if rst= '0' then
```

```
                    current_state <= idle;
                    for i in 7 downto 0 loop
                          dout(i) <= '0';
                    end loop;
            elsif rising_edge(CLK) then

                    case current_state is
                    when idle =>
                    if rx = '0' then
                          next_state <= s0;
                    else
                          next_state <= idle;
                          end if;
                    when s0 =>
                          next_state <= s1;
                          databuffer(0) <= rx;
                    when s1 =>
                          next_state <= s2;
                          databuffer(1) <= rx;
                    when s2 =>
                          next_state <= s3;
                          databuffer(2) <= rx;
                    when s3 =>
                          next_state <= s4;
                          databuffer(3) <= rx;
                    when s4 =>
                          next_state <= s5;
                          databuffer(4) <= rx;
                    when s5 =>
                          next_state <= s6;
                          databuffer(5) <= rx;
                    when s6 =>
                          next_state <= s7;
                          databuffer(6) <= rx;
                    when s7 =>
                          next_state <= stop;
                          databuffer(7) <= rx;
                    when stop =>
                          if rx = '0' then
                                next_state <= s0;
                          else
                                next_state <= idle;
                                end if;
                          dout <= databuffer;
                    end case;
                    current_state <= next_state;
```

```
            end if;
        end process;
    END;
```

## 7.6 通用串行总线

近几年来，通用串行总线（Universal Serial Bus, USB）协议在计算机和电子工业领域变得日益普及。USB协议支持多种数据率，从低速的10~100kbit/s，到高速的400Mbit/s。原则上，创建直连USB总线的FPGA接口是可能的。与低速率的连接不同的是，USB需要串行总线具有精确的电压匹配和阻抗匹配。例如，低速率连接需要用2.8V表示1，0.3V表示0，而高速总线信号的电压要求只有400mV，这两种情况下都需要接末端电阻。

但是，在实际应用中，比较普遍的做法是使用一个简单的接口芯片与FPGA协同工作，接口芯片只负责处理所有的模拟信号，并使用UART类接口同FPGA直接相连。Silicon Labs CP2101就是这样的接口芯片，它以基本的USB连接器引脚（差分数据线、电源线和地线）为输入，输出了基本的串行数据传送引脚。这一器件的框图如图7-6所示。




图7-6 USB收发器芯片CP2101

93

这一器件的引脚意义非常清晰，如下表所示。

| | |
|---|---|
| nRST | 器件的复位引脚，低电平有效 |
| SUSPEND | 这一引脚表示USB器件是否处于挂起状态高电平有效 |
| nSUSPEND | SUSPEND引脚的反相输出，低电平有效 |
| RI | 报警指示 |
| DCD | 数据载波检测，表示数据在USB线上，低电平有效 |
| DTR | 数据传送检测，表示数据线准备好传输数据，低电平有效 |
| DSR | 数字声音重建 |
| TXD | 异步数据传送线 |
| RXD | 异步数据接收线 |
| RTS | 清除接收，低电平有效 |
| CTS | 清除发送，低电平有效 |

这个串行接口的基本操作是从TXD和RXD（数据）线的使用开始的。如果配置是一个没有握手信号的空（NULL）调制解调器（modem），只使用发送线TXD和接收线RXD也是可以的。

如果发送数据前要检查线路是否空闲，可以使用RTS信号（低电平有效）。如果线路已经准备好，那么CTS信号线将变为低电平，表示可以发送数据。这样定义的基本发送方案，一旦接收器的信号变低，发送器就可能用任意速率发送数据，因为它假定接收器可以处理任意速率。

使用DTR线可以使协议具有更强大的能力。这个信号用来通知连接的另一端，已经准备好接收数据。DCD线没有在连接中直接使用，但是它表示在设备之间存在有效的通信连接。

94

我们可以为这种通信连接开发一个VHDL模型，这个模型的复杂度比同实际系统中的硬件进行通信的复杂度还要大，不过我们先从一个简单的模板开始：

```
Entity serial_handler is
      Port(
            Clk : in std_logic;
            Nrst : in std_logic;
            Data_in : in std_logic;
            Data_out : out std_logic;
            TXD : out std_logic;
            RXD : in std_logic
      );
End entity serial_handler;
```

在最初的模型中，有一个时钟和复位信号、两个同步侧的数据连接，还有异步数据通信连线TXD和RXD。我们将它们合成为一个简单的构造体，既可以采样数据线，又可以使用同步模型将数据传输给一个中间变量：

```
Architecture basic of serial_handler is
Begin
      p1 : process (clk)
      Begin
            If rising_edge(clk) then
                  Rxd_int <= rxd;
            End if;
      End process p1;
End architecture basic;
```

我们可以将这个模型进行扩展，让它也可以处理发送侧的事务，如下所示：

```
Architecture basic of serial_handler is
Begin
      p1 : process (clk)
      Begin
            If rising_edge(clk) then
                  Data_out <= rxd;
                  Txd <= data_in;
            End if;
      End process p1;
End architecture basic;
```

这个实体等价于一个空的调制解调器结构。如果我们希望增加DTR（即设备已经准备好

接收数据）提示，可以将它增加到实体的端口列表中。如果使用DTR信号的语句有效，发出接收数据： 95

```
Entity serial_handler is
      Port(
            Clk : in std_logic;
            Nrst : in std_logic;
            Data_in : in std_logic;
            Data_out : out std_logic;
            DTR : in std_logic;
            TXD : out std_logic;
            RXD : in std_logic
      );
End entity serial_handler;
Architecture serial_dtr of serial_handler is
Begin
      p1 : process (clk)
      Begin
            If rising_edge(clk) then
            If DTR = '0' then
                  Data_out <= rxd;
            End if;
            Txd <= data_in;
      End if;
    End process p1;
End architecture basic;
```

使用这种方法，我们就可以将这个串行收发器进行扩展，使它按我们的要求合并一些调制解调器通信连接协议。

## 7.7 小结

本章介绍了多种串行通信编解码方案，同时也回顾了使用RS-232和USB接口的实际方法。但是，对这一主题的阐述可以有多种不同的方案。实际上，要介绍完整的USB协议就需要一本书。 96

# 第8章　数字滤波器

## 8.1　引言

一个电子系统若想与“现实世界”进行接口，必须具备一个重要的功能，那就是处理数字域中采样数据的能力。这通常称为采样数据系统（Sampled Data Systems，SDS），或者称为Z域操作。大多数工程师都熟悉拉普拉斯或者S域中的滤波器运算，S域中一个连续的函数定义了滤波器的特性，而这里所说的Z域运算则是数字域中的等效滤波器运算。

例如，图8-1中所示的模拟RC电路。这是一个低通滤波函数，可以用拉普拉斯符号表示。

等效的S域（拉普拉斯）函数如下：

$$L(s)=\frac{1}{1+sCR}$$

这个函数是一个低通滤波器，因为拉普拉斯（Laplace）算子$s$等价于$j\omega$，其中$\omega=2\pi f$（$f$为频率）。如果$f$等于0（即直流），那么增益为1，但是如果$sCR$的值等于1，那么增益为0.5（即−3 dB）。这是一个经典的低通滤波器截止频率。

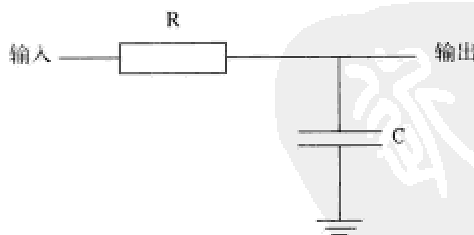





图8-1　模拟电路中的RC滤波器

在数字域中，拉普拉斯算子$s$被Z替代。$Z^{-1}$实际上是一个延迟操作符，数字域（Z域）中与拉普拉斯滤波器类似的函数也可以构造出来。

滤波器的设计技术有许多种，其中很多都超出了本书的范围，但是介绍一些实践中使用的基本技术，并用实例进行说明有助于读者更好地掌握本书的内容。

本章以下部分将涵盖基本技术的介绍，以及如何在FPGA中用VHDL实现这些基本技术的实例说明等内容。

## 8.2　S域到Z域的变换

将滤波器的等式从S域变换到Z域，所用的方法称为“双线性变换（bilinear transform）”。实际上就是用Z域的方法表示S域的等式。基本的变换方法是用Z域的等效符号代替S域中的每一个算子$s$，然后重新将等式化简为最简形式。这一变换之所以称为双线性，是因为变换式中的分子和分母都是线性的。

$$s=\frac{z-1}{z+1}$$

下面举一个简单的例子，变换一个二阶滤波器到Z域中：

$$H(s)=\frac{1}{s^2+2s+1}$$

用$(z-1)/(z+1)$代替s

$$H(z)=\frac{1}{\left(\frac{(z-1)}{(z+1)}\right)^2+\frac{2(z-1)}{(z+1)}+1}$$

$$H(z)=\frac{(z+1)^2}{(z-1)^2+(z-1)(z+1)+(z+1)^2}$$

$$H(z)=\frac{z^2+2z+1}{3z^2+1}$$

98

现在，$H(z)$就是针对输入$X(z)$的输出$Y(z)$，所以可以使用这个关系描述Z域中的等式：

$$H(z)=\frac{z^2+2z+1}{3z^2+1}$$

$$\frac{Y(z)}{X(z)}=\frac{z^2+2z+1}{3z^2+1}$$

$$3z^2Y(z)+Y(z)=z^2X(z)+2zX(z)+X(z)$$

然后可以利用延迟操作符将这一等式改写为一种序列形式的表达式（$z$为一级延迟，$z^2$为二级延迟，依此类推），得到的结果如下：

$$3z^2Y(z)+Y(z)=z^2X(z)+2zX(z)+X(z)$$

$$3y(n+2)+y(n)=x(n+2)+2x(n+1)+x(n)$$

这一等式非常有用，因为现在就可以用延迟的形式来表示Z域中的等式了，最后一步是用前一个单位时刻的值表示当前输出$y(n)$，只要在每个元素中将延迟相应前推若干即可（本例中为2）：

$$3y(n+2)+y(n)=x(x+2)+2x(n+1)+x(n)$$

$$3y(n)+y(n-2)=x(n)+2x(n-1)+x(n-2)$$

$$y(n)+\frac{1}{3}y(n-2)=\frac{1}{3}x(n)+\frac{2}{3}x(n-1)+\frac{1}{3}x(n-2)$$

$$y(n)=\frac{1}{3}x(n)+\frac{2}{3}x(n-1)+\frac{1}{3}x(n-2)-\frac{1}{3}y(n-2)$$

最后还要注意保证设计的频率（例如低通截止频率）要正确。频率在S域和Z域中是不同的，即使在双线性变换之后也是如此。实际设计中，所期望的数字域频率必须利用预变换技术（pre-warping）转换为等价的S域频率。这一步骤将频率从一个域中转换到另一个域中，使用的转换公式如下所示：

$$\omega_c=\tan\left(\frac{\Omega_c T}{2}\right)$$

式中$\Omega_c$为数字域频率，$T$为Z域系统的采样周期，$\omega_c$为模拟域计算用的频率。

一旦得到了Z域的表达式，我们又如何将其转换为实际的设计呢？下一节我们将解释实现的过程。

99

## 8.3 用VHDL实现Z域的函数

### 8.3.1 引言

Z域中的函数本质上是时间域内的数字函数，因为它们是离散的。这些函数在幅度上也是离散的，因为在实际的硬件系统中，都是用固定位数来定义变量或者信号的，不管是整数、有符号数、定点数还是浮点数，对信号而言它们的精度总是有限的。为了简单并且容易理解，本章后面都假定使用有符号算法。这样也就基本定义了系统中使用的位数。如果使用8比特，符号位1比特，可表示的范围是−128～+127。

### 8.3.2 增益模块

第一个主要的Z域模块是增益模块。这个模块有一个有符号输入和一个有符号输出，还有一个参数是增益因子。增益因子可以是整数也可以是有符号数。Z域增益模块的VHDL模型如下列代码所示：

```
Library ieee;
Use ieee.numeric_std.all;

Entity zgain is
      Generic ( n : integer := 8;
              gain : signed
      );
      Port (
            Zin : in signed (n-1 downto 0);
            Zout : out signed (n-1 downto 0)
      );
End entity zgain;

Architecture zdomain of zgain is
Begin
      p1 : process(zin)
            variable product : signed (2*n-1 downto 0);
      begin
            product := zin * gain;
            zout <= product (n-1 downto 0);
      end process p1;
End architecture zdomain;
```

现在，我们用一个简单的测试平台来测试这个模块，让输入递增，然后观察输出的变化：

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity tb is
end entity tb;
```

```
architecture testbench of tb is

       signal clk : std_logic := '0';
       signal dir : std_logic := '0';
       signal zin : signed (7 downto 0):= X"00";
       signal zout : signed (7 downto 0):= X"00";

       component zgain
       generic (
              n : integer := 8;
              gain : signed := X"02"
       );
       port (
             signal zin : in signed(n-1 downto 0);
             signal zout : out signed(n-1 downto 0)
       );
       end component;
       for all : zgain use entity work.zgain;

begin
       clk <= not clk after 1 us;

       DUT : zgain generic map ( 8, X"02" ) port map
         (zin, zout);

       p1 : process (clk)
       begin
             zin <= zin + 1;
       end process p1;
end architecture testbench;
```

很明显，这个模块没有错误检查和范围检查逻辑，这一方法存在的问题是溢出。例如，如果输入64，增益因子设为2，那么结果将是128，但是因为最高位是符号位，所以结果将显示为−128！这类问题在这种简单的模型中是很明显的，所以在设计测试平台时必须加以小心，确保在模型中有足够的检查逻辑。

### 8.3.3 和与差

用同样的方法可以创建和与差的计算模型，它们也是Z域中的基本构造块。求和的VHDL模型如下所示：

```
Library ieee;
Use ieee.numeric_std.all;
Entity zsum is
       Generic ( n : integer := 8
       );
       Port (
             Zin1 : in signed (n-1 downto 0);
```

101

```
            Zin2 : in signed (n-1 downto 0);
            Zout : out signed (n-1 downto 0)
      );
End entity zsum;

Architecture zdomain of zsum is
Begin
      p1 : process(zin)
           variable zsum : signed (2*n-1 downto 0);
      begin
           zsum := zin1 + zin2;
           zout <= zsum (n-1 downto 0);
      end process p1;
End architecture zdomain;
```

尽管也存在溢出的可能，但是这些模型的内部变量宽度是需要宽度的两倍，这样就可以避免任何内部可能出现的溢出了，实际上检查是在最终为输出赋值之前进行的，以确保数据的正确性。求差值的模型几乎一模一样，唯一的区别是计算zin1和zin2的差值。

### 8.3.4 除法模型

在Z域中，对数字进行比例缩放操作的模型是除2模型。这一模型将当前输入值右移1位，即进行了除2操作。可以很容易地将这个模型扩展为右移任意位数，这个简单的模型本身是非常有用的。VHDL代码中使用了逻辑右移操作符SRL，向右移位时丢掉了最低有效位，并在最高有效位补0。相应的VHDL代码如下所示：

```
zout <= zin srl 1;
```

用任意的整数代替移位单位，即可得到特定位数的右移操作。例如，右移3位（相当于除以8）的操作可以用以下代码实现：

102

```
zout <= zin srl 3;
```

完整的除2模型如下：

```
Library ieee;
Use ieee.numeric_std.all;

Entity zdiv2 is
      Generic ( n : integer := 8
      );
      Port (
            Zin : in signed (n-1 downto 0);
            Zout : out signed (n-1 downto 0)
      );
End entity zdiv2;

Architecture zdomain of zdiv2 is
Begin
      zout <= zin srl 1;
End architecture zdomain;
```

为了测试这个模型，我们写一个简单的测试电路，将输入进行递增，代码如下：

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity tb is
end entity tb;

architecture testbench of tb is
      signal clk : std_logic := '0';
      signal dir : std_logic := '0';
      signal zin : signed (7 downto 0) := X"00";
      signal zout : signed (7 downto 0) := X"00";

      component zdiv2
      generic (
            n : integer := 8
      );
      port (
            signal zin : in signed (n-1 downto 0);
            signal zout : out signed (n-1 downto 0)
      );
      end component;
      for all : zdiv2 use entity work.zdiv2;
begin
      clk <= not clk after 1 us;

      DUT : zdiv2 generic map (8) port map (zin, zout);

      p1 : process (clk)
      begin
            zin <= zin + 1;
      end process p1;
end architecture testbench;
```

103

回顾一下这个模型的行为，如果输入为X"03"（十进制数3），二进制为00000011，将这个数右移一位，得到的结果是00000001（X"01"或十进制数1），也就是说，这一操作总是向下取整。显然，这可能会损失精度，而且是向下取整，这也说明了在更复杂的电路中这一操作符是如何处理数字的。

## 8.3.5 单位延迟模型

最后一个基本模型是单位延迟模型zdelay。这一模型具有时钟输入clk，为std_logic类型信号，这样可以简单地与标准数字控制单元进行接口。输出就是简单地将输入延迟了一个时钟周期。

```
Library ieee;
use ieee.std_logic_1164.all;
```

```
Use ieee.numeric_std.all;

Entity zdelay is
      Generic ( n : integer := 8 );
      Port (
            clk : in std_logic;
            Zin : in signed (n-1 downto 0);
            Zout : out signed (n-1 downto 0) := (others
              => '0')
      );
End entity zdelay;

Architecture zdomain of zdelay is
      signal lastzin : signed (n-1 downto 0) := (others
        => '0');
Begin
      p1 : process(clk)
      begin
            if rising_edge(clk) then
                  zout <= lastzin;
                  lastzin <= zin;
            end if;
      end process p1;
End architecture zdomain;
```

注意，输出zout在初始状态下被置为全0，否则“无关”条件可能导致在整个模型中传播。

104

## 8.4 基本低通滤波器模型

将上述模型放在一起，构成一个新的模型，就可以实现需要的基本滤波器，并且要注意在实际应用中确保对溢出错误的检查。

为了说明这一点，我们用方块图来表示一个简单的低通滤波器，如图8-2所示。

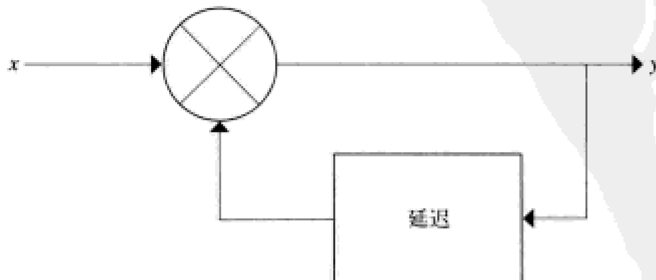



图8-2 简单Z域低通滤波器

可以使用一些前面已经为sum与delay模块设立的独立模型创建一个简单的测试电路，并使输入激励改变一次，然后观察滤波器对这些测试激励的响应。（显然，在这个例子中，滤波器具有统一的增益，并且是正反馈，所以为了保证滤波器的行为正确，我们在sum模块的

两个输入端使用除2模型zdiv2，以使两端都具有0.5的增益。这些在图8-2中并没有显示出来）。最后的VHDL模型如下所示（注意zdiv2模型的使用）：

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity tb is
end entity tb;

architecture testbench of tb is
        signal clk : std_logic := '0';
        signal x : signed (7 downto 0):= X"00";
        signal y : signed (7 downto 0):= X"00";
        signal y1 : signed (7 downto 0):= X"00";
        signal yd : signed (7 downto 0):= X"00";
        signal yd2 : signed (7 downto 0):= X"00";
        signal x2 : signed (7 downto 0):= X"00";
component zsum
generic (
        n : integer := 8
);
port (
        signal zin1 : in signed(n-1 downto 0);
        signal zin2 : in signed(n-1 downto 0);
        signal zout : out signed(n-1 downto 0)
);
end component;
for all : zsum use entity work.zsum;

component zdiff
generic (
        n : integer := 8
);
port (
        signal zin1 : in signed(n-1 downto 0);
        signal zin2 : in signed(n-1 downto 0);
        signal zout : out signed(n-1 downto 0)
);
end component;
for all : zdiff use entity work.zdiff;
     component zdiv2
     generic (
          n : integer := 8
     );
     port (
```

105

```
            signal zin : in signed(n-1 downto 0);
            signal zout : out signed(n-1 downto 0)
        );
        end component;
        for all : zdiv2 use entity work.zdiv2;

        component zdelay
        generic (
            n : integer := 8
        );
        port (
            signal clk : in std_logic;
            signal zin : in signed(n-1 downto 0);
            signal zout : out signed(n-1 downto 0)
        );
        end component;
        for all : zdelay use entity work.zdelay;
begin
        clk <= not clk after 1 us;

        GAIN1 : zdiv2 generic map (8) port map (x, x2);
        GAIN2 : zdiv2 generic map (8) port map (yd, yd2);
        SUM1 : zsum generic map (8) port map (x2, yd2,
          y);
        D1 : zdelay generic map (8) port map (clk, y,
          yd) ;
        x <= X"00", X"0F" after 10 us;
end architecture testbench;
```

106

测试电路在10μs之后将输入从X"00"改变到X"0F"。结果引起滤波器出现响应。我们将结果显示在图8-3中，输出用十六进制和模拟方式显示。

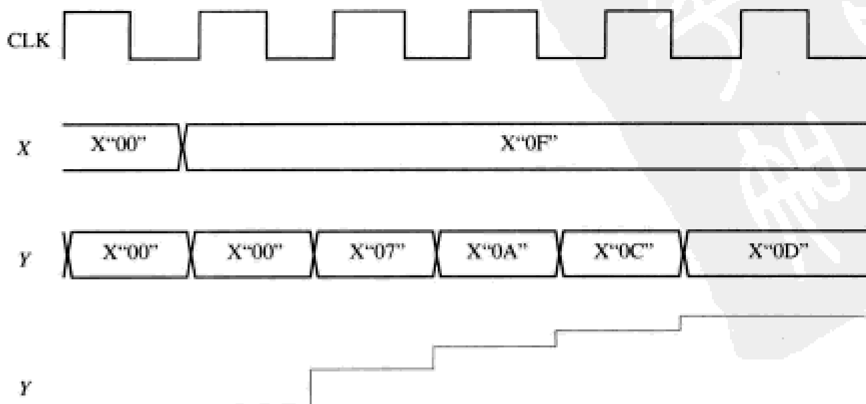



图8-3　基本低通滤波器仿真波形

有意思的是，用zdiv2函数对结果产生了一些影响。对于输入0F（二进制数为00001111），

除以2就丢失了最低有效位（LSB），结果得到00000111（7），将这个数作为sum模块的输入，与输出的除法结果相加得到14，这是滤波器最大可能的输出。滤波器实际的输出却是X"0D"二进制为00001101，这比理论最大值X"0F"小2，这也说明了用“粗放的”近似方法代替定点或浮点方法给数字处理工作带来的实际困难。另一方面，这也是一种用VHDL实现基本滤波器的简单而有效的方法。

本书后面的部分将讨论定点数和浮点数的使用问题，还要讨论精确计算以及滤波器设计中使用的乘法器的使用。这两个领域要求较高的计算精度，所有这些方法都有可能用到。然而，有些情况下不能使用这些高级的技术，尤其是在FPGA空间比较紧张的时候，这种情况下就必须使用本章介绍的简单方法。

有许多教科书介绍数字滤波器设计的一些更高级的主题，这些内容已经超过了本书的范围，但是在这里介绍一下当今常用的两种主要数字滤波器的一些关键概念还是有用的。这两种滤波器类型包括递归型（或称为无限脉冲响应，IIR）滤波器和非递归型（有限脉冲响应，FIR）滤波器。 107

## 8.5 FIR滤波器

FIR滤波器的特征是，只使用输入信号的延迟部分来对输入进行滤波，并产生输出。例如，使用如下的通用FIR滤波器，可以看到输出是输入信号的一系列延迟、比例缩放部分的函数：

$$y = \sum_{i=0}^{n} A_i x[i]$$

式中的$A_i$是输入信号的第$i$个延迟对应的比例因子。我们可以在图8-4中将这个滤波器表示出来。使用本章前面描述的构造块就可以实现这个模型，例如增益（gain）、除法（division）、和（sum）以及延迟（delay）等模块。在前面的部分已经注意到，重要的一点是要保证，为 108

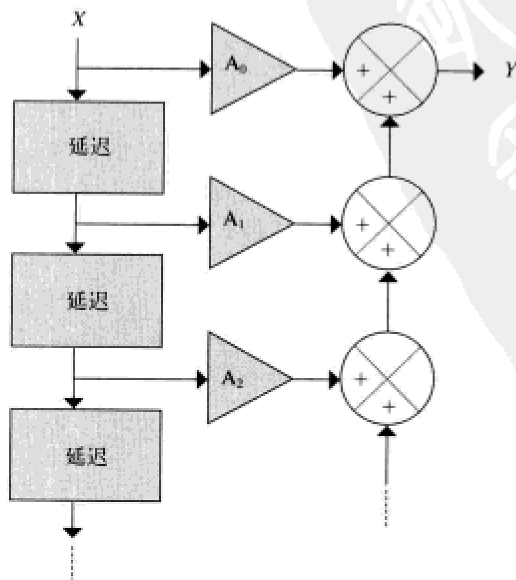



图8-4 有限脉冲响应（FIR）滤波器

了实现更高精度的滤波器，必须使用定点数和浮点数算法，而且在大多数情况下，使用更高精度的乘法器是更可取的。

## 8.6 IIR滤波器

IIR滤波器的特征是，使用输入信号的延迟部分以及输出信号的反馈和延迟对输入进行滤波，并产生输出。例如，使用如下的通用IIR滤波器，可以看到输出是输入信号和输出信号的一系列延迟、比例缩放部分的函数：

$$y = \sum_{i=0}^{n} \frac{A_i x[i]}{B_i y[i]}$$

式中的$A_i$是输入信号的第$i$个延迟对应的比例因子，$B_i$是输出信号的第$i$个延迟对应的比例因子。很显然，这一公式与前面提到的FIR滤波器公式非常相似，也可以使用同样的基本元件组成。本章开始时提到的一个简单例子，实际上可以看作一个简单的一阶（单级延迟）IIR滤波器，不使用输入信号的延迟，而使用输出信号的单级延迟。

## 8.7 小结

本章介绍了用VHDL实现基本数字滤波器的概念，并给出了一些在FPGA平台上用构造块方法和结构化方法实现滤波器的例子。另外，也介绍了FIR和IIR滤波器，读者可以藉此为自己的应用开发一些合适的滤波器。

109



# 第9章　安全系统

## 9.1　块加密简介

数据加密标准（DES）是一种对称的“块加密”。流加密是每次对一或多比特数据流进行操作，而块加密则是每次都对一个完整的数据块进行操作，并产生大小相等的密文块。DES作为一种块加密，其数据块的大小为64比特。DES使用64比特的密钥，但每8比特中有1比特为奇偶校验码，因此实际的密钥为56比特。DES与其他的块加密有共同之处，那就是基于一种称为“费斯特尔格子（FEISTEL LATTICE）”的结构，所以这里先介绍一下这种结构的工作原理。

## 9.2　费斯特尔格子的结构

块加密对一个$n$比特的明文数据块进行加密，产生$n$比特的密文输出。由于加密算法可能是可逆的（也就是说，可能要解密），因此在明文数据块和密文数据块之间必须是一个单一映射。这也可以称为“非退化变换”。例如图9-1所示的变换。

| 可逆 | |
|---|---|
| 明文 | 密文 |
| 00 | 11 |
| 01 | 10 |
| 10 | 00 |
| 11 | 10 |

| 不可逆 | |
|---|---|
| 明文 | 密文 |
| 00 | 11 |
| 10 | 10 |
| 10 | 10 |
| 11 | 10 |

图9-1　可逆与不可逆变换

显然，这是一个置换加密算法，很容易被简单的密码分析学中使用的标准统计分析技术所攻击（例如频率分析）。随着数据块大小的不断增加，这种实现方法变得越来越缺乏可行性。这种方法在具体实现时有一个明显的困难，那就是所需的变换次数按比特数$n$增加。对于一个$n$比特的置换型块加密，密钥大小为$n \times 2^n$。例如，当$n$=64时，密钥大小就是$64 \times 2^{64} \approx 10^{21}$。

110

为了解决这个复杂的问题，费斯特尔提出了一种方法，称为“产品加密”，这种方法将几个简单的步骤结合起来得到了比组件加密（component cipher）所使用的任何方法都要高的安全特性。费斯特尔的方法依赖于以下两种功能的交替使用：

- 扩散（diffusion）
- 混合（confusion）

这两个概念来源于香农开发出来的方法，现在使用的大多数标准块加密算法普遍使用了这两个概念。香农的目标是定义出不容易被统计分析攻击的加密函数。他提出两种方法可以削弱统计分析找到原始信息的能力：扩散与混合。

在扩散操作中，将明文的统计结构扩散到密文的全部长期统计特性中。这可以通过令明文的每一比特都影响密文的多个比特的值来实现。举一个实际的例子，在密文中增加一些字母，以使每一个字母的出现频率相同，而不管原始信息是什么。在二进制加密算法中，这一技术使用多次排列的方法，使密文的每一比特都受明文的多个比特影响。

每一个明文块都要变换为一个密文块，并且与密钥有关。混合的目的是使密文块与密钥之间的关系尽可能复杂，以降低获知密钥的可能性。这就要求有一个复杂的置换算法作为一种不保护密钥的线性置换。

扩散和混合都是一个成功的块加密设计的基石。

这些需求的结果就是费斯特尔格子（如图9-2所示）。这是加密算法（如DES）的基本结构。

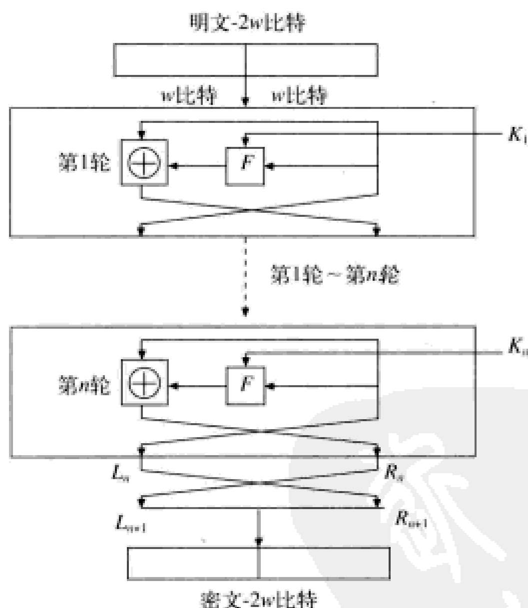



图9-2 费斯特尔格子结构

加密算法的输入是明文（长度为2$w$比特）和密钥$K$。首先将明文分为两部分$L$和$R$，长度各为$w$比特，然后数据通过$n$轮的处理，最后再重新组合起来得到密文。每一轮的输入$L_{i-1}$和$R_{i-1}$都来自上一轮的输出，子密钥$K_i$由初始密钥$K$扩展而来。每一轮变换的结构都完全相同。左边一半数据需要执行一次置换操作。这就要求对右边一半数据执行一次"轮函数$F$"，然后与左边一半数据进行异或，最后再将数据的两个部分交换。

111

费斯特尔网络的实现有以下一些重要的参数。

- 数据块大小。数据块越大通常意味着安全性更高，但是速度会降低。尽管高级加密标准（Advanced Encryption Standards，AES）使用128比特数据块，但是最常使用的还是64比特的数据块，其比较折中较为合理。
- 密钥大小。与数据块大小具有同样的折中情况。一般来讲，现在使用64比特密钥已经不够了，最好选用128比特密钥。
- 轮数。每一轮都可以增加安全性。单轮是不够的，一般的标准轮数是16。

❑ 子密钥生成。子密钥生成算法越复杂，整个系统的安全性也就越高。

❑ 轮函数。轮函数越复杂就意味着破解密码的难度越大。

112

## 9.3 数据加密标准

### 9.3.1 引言

美国国家标准与技术局（National Institute of Standards and Technology，NIST）于1977年采用DES作为联邦信息处理标准（Federal Information Processing Standards）46号，即FIPS PUB 46。

正如前面提到的，这一算法对64比特明文进行处理，密钥长度为56比特。到1999年，美国国家标准与技术局颁布法令，宣布DES不再安全，只能用于以前遗留下来的系统中，此后应该使用三重DES。后面将会说明，DES已经被AES所代替。

DES的粗略结构（整体结构）如图9-3所示。

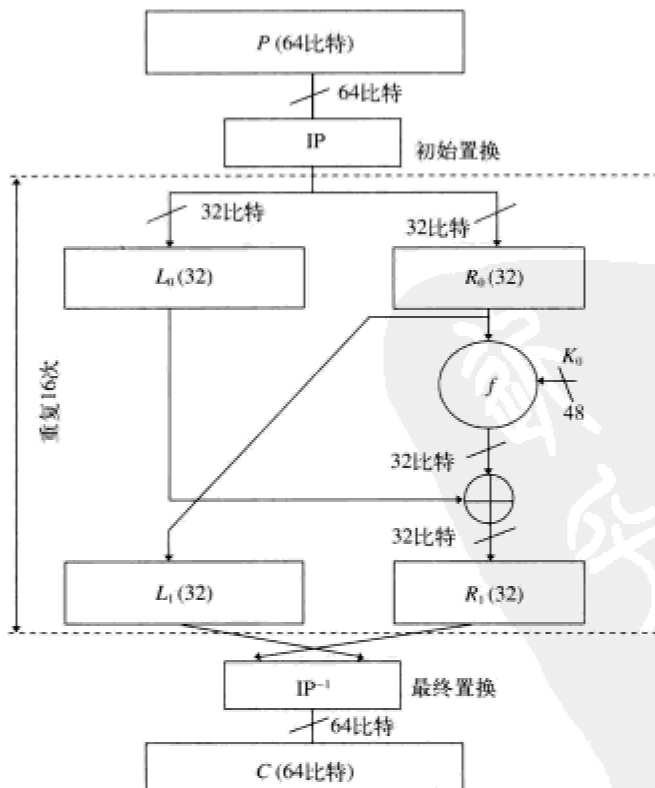



图9-3 DES粗略结构

中间部分（主循环部分）称为精细结构，加密操作的细节发生在这里。该精细结构的详细情况如图9-4所示。

DES精细结构中包含以下几个重要的功能模块。

❑ 初始置换：映射64×64比特。

❑ 密钥转换：密钥循环左移$A(i)$位［$A(i)$已知并固定］。

113

- 压缩置换：将56比特输入映射为48比特输出。
- 扩展置换：通过复制16比特输入，将32比特数据混合并映射（这两种操作都是确定的）为48比特。这使得扩散更快。

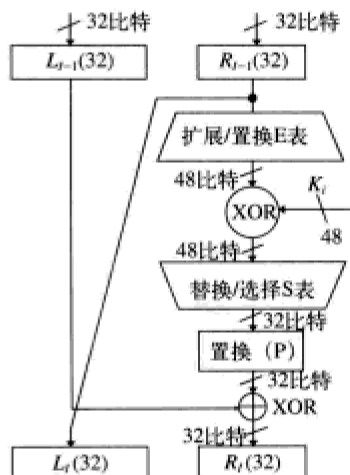


图9-4　DES精细结构

算法中另一个重要的部分是替换，或称为S盒。密码学中的非线性特征非常重要。在DES算法中，共有8个S盒，每一个S盒包含4种不同（确定的）的4:4输入映射。可以通过扩展置换中产生的额外位来选择这些不同的映射。S盒的结构如图9-5所示。

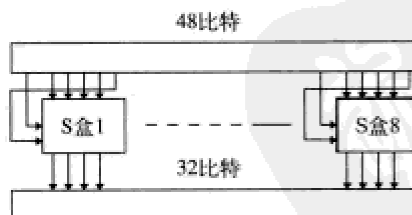


图9-5　S盒结构

114　DES中最后一部分是密钥生成结构，用于产生每一轮中独立的密钥，如图9-6所示。

剩下的功能模块是初始置换和最终置换。初始置换（P盒）是一个32:32的固定位置换。最终置换是初始置换的逆过程。初始置换使用下表定义。

| 58 | 50 | 42 | 34 | 26 | 18 | 10 | 2 |
|---|---|---|---|---|---|---|---|
| 60 | 52 | 44 | 36 | 28 | 20 | 12 | 4 |
| 62 | 54 | 46 | 38 | 30 | 22 | 14 | 6 |
| 64 | 56 | 48 | 40 | 32 | 24 | 16 | 8 |
| 57 | 49 | 41 | 33 | 25 | 17 | 9 | 1 |
| 59 | 51 | 43 | 35 | 27 | 19 | 11 | 3 |
| 61 | 53 | 45 | 37 | 29 | 21 | 13 | 5 |
| 63 | 55 | 47 | 39 | 31 | 23 | 15 | 7 |

最终置换只要将初始置换反序操作即可。

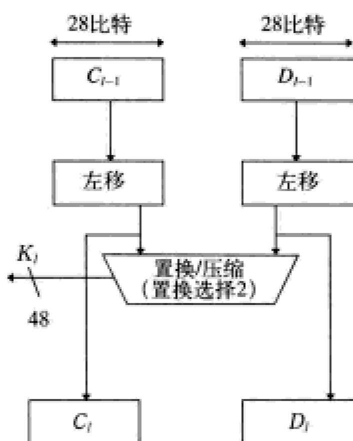



图9-6 DES轮密钥生成

## 9.3.2 DES的VHDL实现

可以使用结构化或者功能化的方法来实现DES。前面已经讨论过，这两种方法各有优缺点，但是DES算法本身具有结构化的特点，所以使用结构化方法能够得到更有效的实现。 [115]

用VHDL实现初始置换需要一个64比特的输入和一个64比特的输出。VHDL实体如下，输入和输出定义为std_logic_vector类型：

```
library ieee;
use ieee.std_logic_1164.all;

entity des_ip is port
        (
              D : in std_logic_vector(1 TO 64);
              Y : out std_logic_vector(1 TO 64)
        );
end des_ip;
```

构造体中，直接根据前面定义的初始置换表将输入赋值给输出即可：

```
architecture behavior of des_ip is
begin
  Y(1)<=D(58);      Y(2)<=D(50);      Y(3)<=D(42);      Y(4)<=D(34);
  Y(5)<=D(26);      Y(6)<=D(18);      Y(7)<=D(10);      Y(8)<=D(2);
  Y(9)<=D(60);      Y(10)<=D(52);     Y(11)<=D(44);     Y(12)<=D(36);
  Y(13)<=D(28);     Y(14)<=D(20);     Y(15)<=D(12);     Y(16)<=D(4);
  Y(17)<=D(62);     Y(18)<=D(54);     Y(19)<=D(46);     Y(20)<=D(38);
  Y(21)<=D(30);     Y(22)<=D(22);     Y(23)<=D(14);     Y(24)<=D(6);
  Y(25)<=D(64);     Y(26)<=D(56);     Y(27)<=D(48);     Y(28)<=D(40);
  Y(29)<=D(32);     Y(30)<=D(24);     Y(31)<=D(16);     Y(32)<=D(8);
  Y(33)<=D(57);     Y(34)<=D(49);     Y(35)<=D(41);     Y(36)<=D(33);
  Y(37)<=D(25);     Y(38)<=D(17);     Y(39)<=D(9);      Y(40)<=D(1);
  Y(41)<=D(59);     Y(42)<=D(51);     Y(43)<=D(43);     Y(44)<=D(35);
```

```
  Y(45)<=D(27);     Y(46)<=D(19);     Y(47)<=D(11);     Y(48)<=D(3);
  Y(49)<=D(61);     Y(50)<=D(53);     Y(51)<=D(45);     Y(52)<=D(37);
  Y(53)<=D(29);     Y(54)<=D(21);     Y(55)<=D(13);     Y(56)<=D(5);
  Y(57)<=D(63);     Y(58)<=D(55);     Y(59)<=D(47);     Y(60)<=D(39);
  Y(61)<=D(31);     Y(62)<=D(23);     Y(63)<=D(15);     Y(64)<=D(7);
end behavior;
```

这一功能是纯粹的组合逻辑，不需要寄存器（也就是说不需要时钟）。不过如果需要的话，也可以在一个过程（process）中这样实现。

116 如前面所示的扩展功能一样，需要将一个32比特的数据扩展为48比特。这就需要一个转换表，如下所示。需要注意的是，转换单元中有复制操作，也就是说只要有32比特输入就可以得到48比特的输出：

```
32    1     2     3     4     5
4     5     6     7     8     9
8     9     10    11    12    13
12    13    14    15    16    17
16    17    18    19    20    21
20    21    22    23    24    25
24    25    26    27    28    29
28    29    30    31    32    1
```

可以使用与初始置换相似的VHDL模型来实现，只不过此时的输入为32比特，输出为48比特。注意，有一部分输入位有重复，这样就完成了扩展功能：

```
library ieee;
use ieee.std_logic_1164.all;

entity des_e is port
       (
            D : in std_logic_vector(1 TO 32);
            Y : out std_logic_vector(1 TO 48)
       );
end des_e;
```

构造体也是根据前面定义的扩展转换表将输入赋值给输出即可：

```
architecture behavior of des_e is
begin
  Y(1)<=D(32);      Y(2)<=D(1);       Y(3)<=D(2);       Y(4)<=D(3);
  Y(5)<=D(4);       Y(6)<=D(5);       Y(7)<=D(4);       Y(8)<=D(5);
  Y(9)<=D(6);       Y(10)<=D(7);      Y(11)<=D(8);      Y(12)<=D(9);
  Y(13)<=D(8);      Y(14)<=D(9);      Y(15)<=D(10);     Y(16)<=D(11);
  Y(17)<=D(12);     Y(18)<=D(13);     Y(19)<=D(12);     Y(20)<=D(13);
  Y(21)<=D(14);     Y(22)<=D(15);     Y(23)<=D(16);     Y(24)<=D(17);
  Y(25)<=D(16);     Y(26)<=D(17);     Y(27)<=D(18);     Y(28)<=D(19);
  Y(29)<=D(20);     Y(30)<=D(21);     Y(31)<=D(20);     Y(32)<=D(21);
  Y(33)<=D(22);     Y(34)<=D(23);     Y(35)<=D(24);     Y(36)<=D(25);
  Y(37)<=D(24);     Y(38)<=D(25);     Y(39)<=D(26);     Y(40)<=D(27);
```

```
  Y(41)<=D(28);      Y(42)<=D(29);      Y(43)<=D(28);      Y(44)<=D(29);
  Y(45)<=D(30);      Y(46)<=D(31);      Y(47)<=D(32);      Y(48)<=D(1);
end behavior;
```

最终置换模块就是图9-4精细结构中的P，位于密钥函数之后。这是一种32比特到32比特的直接置换。比特映射表如下表所示： 117

```
16    7     20    21
29    12    28    17
1     15    23    26
5     18    31    10
2     8     24    14
32    27    3     9
19    13    30    6
22    11    4     25
```

实现方法与前面的扩展和置换完全相同，VHDL代码如下：

```
library ieee;
use ieee.std_logic_1164.all;

entity des_p is port
       (
           D : in std_logic_vector(1 TO 32);
           Y : out std_logic_vector(1 TO 32)
       );
end des_p;
```

构造体也是按照前面定义的比特映射表直接将输入赋值给输出即可：

```
architecture behavior of des_p is
begin
  Y(1)<=D(16);     Y(2)<=D(7);      Y(3)<=D(20);     Y(4)<=D(21);
  Y(5)<=D(29);     Y(6)<=D(12);     Y(7)<=D(28);     Y(8)<=D(17);
  Y(9)<=D(1);      Y(10)<=D(15);    Y(11)<=D(23);    Y(12)<=D(26);
  Y(13)<=D(5);     Y(14)<=D(18);    Y(15)<=D(31);    Y(16)<=D(10);
  Y(17)<=D(2);     Y(18)<=D(8);     Y(19)<=D(24);    Y(20)<=D(14);
  Y(21)<=D(32);    Y(22)<=D(27);    Y(23)<=D(3);     Y(24)<=D(9);
  Y(25)<=D(19);    Y(26)<=D(13);    Y(27)<=D(30);    Y(28)<=D(6);
  Y(29)<=D(22);    Y(30)<=D(11);    Y(31)<=D(4);     Y(32)<=D(25);
end behavior;
```

DES算法的非线性部分是S盒。这是一种6比特到4比特的转换，目的是将DES中*F*函数扩展出来的48比特数据减少为下一轮需要的32比特数据。行号和列号来自输入给S盒的数据。S盒的输入数据为6比特二进制数。行号由$2b_1+b_6$构成，列号由$b_2b_3b_4b_5$构成。例如，S（011011）的行号为01（即1），列号为1101（即13），则S8将返回值1110（即14）。 118

因此，可以用以下的VHDL代码构造出基本的S盒实体：

```
Library ieee;
```

```
Use ieee.std_logic_1164.all;
Entity des_sbox is
      Port (
            D : in std_logic_vector (1 to 6);
            Y : out std_logic_vector (1 to 4)
      );
End entity des_sbox;
```

一种方法是根据输入数据*D*定义行号和列号，然后使用查找表或者用真值表的逻辑化简方法计算出输出*Y*。其构造体类似于如下的形式：

```
Architecture behaviour of sbox is
      Signal r : std_logic_vector (1 to 2);
      Signal c : std_logic_vector (3 to 6);
Begin
      R <= d ( 1 to 2);
      C <= d (3 to 6 );
      -- The look up table or logic goes here
End;
```

另一种方法是根据输入数据*D*定义一个简单的查找表，其地址唯一，而输出*Y*则存在存储器中，这与只读存储器（ROM）完全相同，所以输入可以定义为无符号整数，用以查找需要的数据。这种情况下，存储器的定义方法与本书介绍的ROM定义方法完全相同。

S盒替换由下表指定，相应的VHDL代码要么使用查找表存储每次替换的地址，要么用逻辑方式译码找出正确的输出。

为了使用这个表，首先选择适当的S盒，然后用两位行地址选择行，用四位列地址选择列。例如，对于S1盒，如果行地址为3（即二进制数11），列地址为10（二进制数为1010），那么输出将是3（二进制数0011）。这可以使用嵌套的case语句实现，代码如下：



```
Case row is
      When 0 =>
             Case column is
                   When 0 => y <= 14;
                   When 1 => y <= 4;
                   ...
             End case;
      When 1 =>
             Case column is
                   ...
             End case
      ...
End case;
```

很明显，这种实现方法相当繁琐，但是却很容易自动生成代码，而且为综合工具执行逻辑优化提供了可能性，这样就比存储器实现方式有更好的效率。

| 行 | 列数 | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] | [10] | [11] | [12] | [13] | [14] | [15] |
| $S_1$ | | | | | | | | | | | | | | | | |
| [0] | 14 | 4 | 13 | 1 | 2 | 15 | 11 | 8 | 3 | 10 | 6 | 12 | 5 | 9 | 0 | 7 |
| [1] | 0 | 15 | 7 | 4 | 14 | 2 | 13 | 1 | 10 | 6 | 12 | 11 | 9 | 5 | 3 | 8 |
| [2] | 4 | 1 | 14 | 8 | 13 | 6 | 2 | 11 | 15 | 12 | 9 | 7 | 3 | 10 | 5 | 0 |
| [3] | 15 | 12 | 8 | 2 | 4 | 9 | 1 | 7 | 5 | 11 | 3 | 14 | 10 | 0 | 6 | 13 |
| $S_2$ | | | | | | | | | | | | | | | | |
| [0] | 15 | 1 | 8 | 14 | 6 | 11 | 3 | 4 | 9 | 7 | 2 | 13 | 12 | 0 | 5 | 10 |
| [1] | 3 | 13 | 4 | 7 | 15 | 2 | 8 | 14 | 12 | 0 | 1 | 10 | 6 | 9 | 11 | 5 |
| [2] | 0 | 14 | 7 | 11 | 10 | 4 | 13 | 1 | 5 | 8 | 12 | 6 | 9 | 3 | 2 | 15 |
| [3] | 13 | 8 | 10 | 1 | 3 | 15 | 4 | 2 | 11 | 6 | 7 | 12 | 0 | 5 | 14 | 9 |
| $S_3$ | | | | | | | | | | | | | | | | |
| [0] | 10 | 0 | 9 | 14 | 6 | 3 | 15 | 5 | 1 | 13 | 12 | 7 | 11 | 4 | 2 | 8 |
| [1] | 13 | 7 | 0 | 9 | 3 | 4 | 6 | 10 | 2 | 8 | 5 | 14 | 12 | 11 | 15 | 1 |
| [2] | 13 | 6 | 4 | 9 | 8 | 15 | 3 | 0 | 11 | 1 | 2 | 12 | 5 | 10 | 14 | 7 |
| [3] | 1 | 10 | 13 | 0 | 6 | 9 | 8 | 7 | 4 | 15 | 14 | 3 | 11 | 5 | 2 | 12 |
| $S_4$ | | | | | | | | | | | | | | | | |
| [0] | 7 | 13 | 14 | 3 | 0 | 6 | 9 | 10 | 1 | 2 | 8 | 5 | 11 | 12 | 4 | 15 |
| [1] | 13 | 8 | 11 | 5 | 6 | 15 | 0 | 3 | 4 | 7 | 2 | 12 | 1 | 10 | 14 | 9 |
| [2] | 10 | 6 | 9 | 0 | 12 | 11 | 7 | 13 | 15 | 1 | 3 | 14 | 5 | 2 | 8 | 4 |
| [3] | 3 | 15 | 0 | 6 | 10 | 1 | 13 | 8 | 9 | 4 | 5 | 11 | 12 | 7 | 2 | 14 |
| $S_5$ | | | | | | | | | | | | | | | | |
| [0] | 2 | 12 | 4 | 1 | 7 | 10 | 11 | 6 | 8 | 5 | 3 | 15 | 13 | 0 | 14 | 9 |
| [1] | 14 | 11 | 2 | 12 | 4 | 7 | 13 | 1 | 5 | 0 | 15 | 10 | 3 | 9 | 8 | 6 |
| [2] | 4 | 2 | 1 | 11 | 10 | 13 | 7 | 8 | 15 | 9 | 12 | 5 | 6 | 3 | 0 | 14 |
| [3] | 11 | 8 | 12 | 7 | 1 | 14 | 2 | 13 | 6 | 15 | 0 | 9 | 10 | 4 | 5 | 3 |
| $S_6$ | | | | | | | | | | | | | | | | |
| [0] | 12 | 1 | 10 | 15 | 9 | 2 | 6 | 8 | 0 | 13 | 3 | 4 | 14 | 7 | 5 | 11 |
| [1] | 10 | 15 | 4 | 2 | 7 | 12 | 9 | 5 | 6 | 1 | 13 | 14 | 0 | 11 | 3 | 8 |
| [2] | 9 | 14 | 15 | 5 | 2 | 8 | 12 | 3 | 7 | 0 | 4 | 10 | 1 | 13 | 11 | 6 |
| [3] | 4 | 3 | 2 | 12 | 9 | 5 | 15 | 10 | 11 | 14 | 1 | 7 | 6 | 0 | 8 | 13 |
| $S_7$ | | | | | | | | | | | | | | | | |
| [0] | 4 | 11 | 2 | 14 | 15 | 0 | 8 | 13 | 3 | 12 | 9 | 7 | 5 | 10 | 6 | 1 |
| [1] | 13 | 0 | 11 | 7 | 4 | 9 | 1 | 10 | 14 | 3 | 5 | 12 | 2 | 15 | 8 | 6 |
| [2] | 1 | 4 | 11 | 13 | 12 | 3 | 7 | 14 | 10 | 15 | 6 | 8 | 0 | 5 | 9 | 2 |
| [3] | 6 | 11 | 13 | 8 | 1 | 4 | 10 | 7 | 9 | 5 | 0 | 15 | 14 | 2 | 3 | 12 |
| $S_8$ | | | | | | | | | | | | | | | | |
| [0] | 13 | 2 | 8 | 4 | 6 | 15 | 11 | 1 | 10 | 9 | 3 | 14 | 5 | 0 | 12 | 7 |
| [1] | 1 | 15 | 13 | 8 | 10 | 3 | 7 | 4 | 12 | 5 | 6 | 11 | 0 | 14 | 9 | 2 |
| [2] | 7 | 11 | 4 | 1 | 9 | 12 | 14 | 2 | 0 | 6 | 10 | 13 | 15 | 3 | 5 | 8 |
| [3] | 2 | 1 | 14 | 7 | 4 | 10 | 8 | 13 | 15 | 12 | 9 | 0 | 3 | 5 | 6 | 11 |

120

### 9.3.3 DES的验证

为了验证DES的实现结果，可以使用一些测试向量以保证处理结果的正确性。测试向量如下。

明文：4E6F772069732074 68652074696D6520 666F7220616C6C20

密文：3FA40E8A984D4815 6A271787AB8883F9 893D51EC4B563B53

这里所用的密钥是0123456789ABCDEF。

每一个十六进制字符由7比特的ASCII码和1比特的附加位表示。

## 9.4 高级加密标准

1997年，NIST发布征集令，要求为非机密官方文件创建一种新的高级加密标准（AES）加密算法。这一征集也要求AES是一种完全公开的、全世界都可以免费使用的加密算法。另外，也要求算法必须按照块密码的要求实现对称密钥，而且至少要支持128比特宽度的块大小，以及128比特、192比特和256比特的密钥长度。

通过公开的竞争，Rijndael算法成为优胜者，并被作为AES标准。Rijndael算法允许密钥大小和块大小为128比特、192比特或者256比特。AES中对密钥长度的规定与Rijndael算法完全相同，但是数据块大小确定为128比特。算法的操作流程与DES相似，每次对一个数据块的操作都包含10轮的混合与扩散操作。每一轮都有单独的密钥，这些密钥都是由总密钥产生出来的。轮的结构如图9-7所示。

121

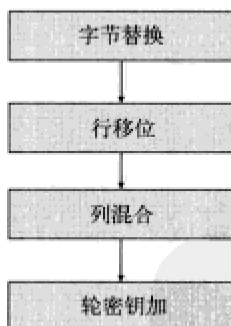



图9-7 AES轮结构

总体的AES结构如图9-8所示。

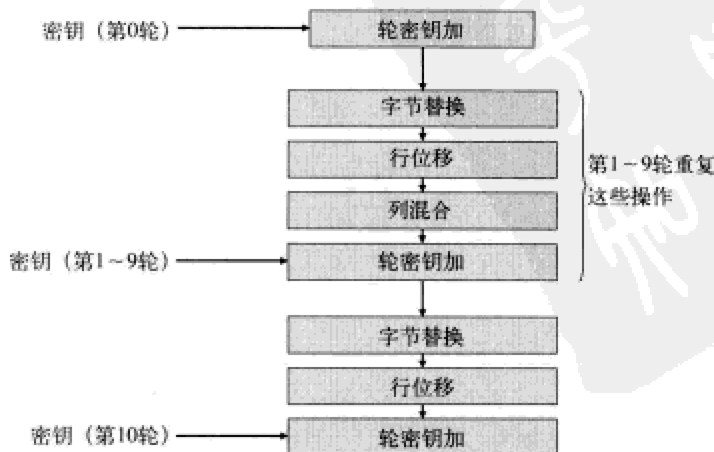



图9-8 AES结构

每一个数据块都包含128比特，这128比特被分成16个8比特字节。以后的每一步操作都是对这些排列成4×4矩阵的字节进行的：

$$\begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix} \tag{9-1}$$

式中，每一个$a_{i,j}$都是一个8比特的字节，并且是GF ($2^8$)域内的一个元素。计算过程遵循Rijindael算法中规定的伽罗华域（Galois Field）规则，例如，在伽罗华域中，加法可以用异或（xor）实现。

乘法更复杂一些，每一个字节都有一个逆元，所以字节与其逆元的乘积为1，如$b \cdot b'$ =00000001（不包含00000000，这个元素的逆元是它本身）。

122

有限域GF($2^8$)的模型由一个8阶的不可约多项式定义，Rijindael算法中使用了如下的生成多项式：

$$X^8+X^4+X^3+1 \tag{9-2}$$

每一轮操作都需要一个特定的数学操作。依次列举如下。

字节替换要求：对每一个输入数据块$a$ (3, 3)，查找字节替换表，用查找到的字节替换原来的字节得到一个新的矩阵$b$ (3,3)。工作的方式是，对输入字节abcdefgh，用abcd作为行地址，efgh作为列地址，找到该位置的字节作为输出：

$$\begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix} \Rightarrow \text{字节替换}$$

$$\Rightarrow \begin{pmatrix} b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} \end{pmatrix} \tag{9-3}$$

完整的字节替换表如下所示。

AES字节替换表

| 099 | 124 | 119 | 123 | 242 | 107 | 111 | 197 | 048 | 001 | 103 | 043 | 254 | 215 | 171 | 118 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 202 | 130 | 201 | 125 | 250 | 089 | 071 | 240 | 173 | 212 | 162 | 175 | 156 | 164 | 114 | 192 |
| 183 | 253 | 147 | 038 | 054 | 063 | 247 | 204 | 052 | 165 | 229 | 241 | 113 | 216 | 049 | 021 |
| 004 | 199 | 035 | 195 | 024 | 150 | 005 | 154 | 007 | 018 | 128 | 226 | 235 | 039 | 178 | 117 |
| 009 | 131 | 044 | 026 | 027 | 110 | 090 | 160 | 082 | 059 | 214 | 179 | 041 | 227 | 047 | 132 |
| 083 | 209 | 000 | 237 | 032 | 252 | 177 | 091 | 106 | 203 | 190 | 057 | 074 | 076 | 088 | 207 |
| 208 | 239 | 170 | 251 | 067 | 077 | 051 | 133 | 069 | 249 | 002 | 127 | 080 | 060 | 159 | 168 |
| 081 | 163 | 064 | 143 | 146 | 157 | 056 | 245 | 188 | 182 | 218 | 033 | 016 | 255 | 243 | 210 |
| 205 | 012 | 019 | 236 | 095 | 151 | 068 | 023 | 196 | 167 | 126 | 061 | 100 | 093 | 025 | 115 |
| 096 | 129 | 079 | 220 | 034 | 042 | 144 | 136 | 070 | 238 | 184 | 020 | 222 | 094 | 011 | 219 |
| 224 | 050 | 058 | 010 | 073 | 006 | 036 | 092 | 194 | 211 | 172 | 098 | 145 | 149 | 228 | 121 |

（续）

| AES字节替换表 | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 231 | 200 | 055 | 109 | 141 | 213 | 078 | 169 | 108 | 086 | 244 | 234 | 101 | 122 | 174 | 008 |
| 186 | 120 | 037 | 046 | 028 | 166 | 180 | 198 | 232 | 221 | 116 | 031 | 075 | 189 | 139 | 138 |
| 112 | 062 | 181 | 102 | 072 | 003 | 246 | 014 | 097 | 053 | 087 | 185 | 134 | 193 | 029 | 158 |
| 225 | 248 | 152 | 017 | 105 | 217 | 142 | 148 | 155 | 030 | 135 | 233 | 206 | 085 | 040 | 223 |
| 140 | 161 | 137 | 013 | 191 | 230 | 066 | 104 | 065 | 153 | 045 | 015 | 176 | 084 | 187 | 022 |

123

举例说明如下。

若输入数据为7A，那么其二进制表示为0111 1010，这样行地址为7（0111），列地址为A（1010）。按照这个地址从字节替换表中就可以得到替换结果：

218=1101 1010=DA

替换过程如下所示：

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 099 | 124 | 119 | 123 | 242 | 107 | 111 | 197 | 048 | 001 | 103 | 043 | 254 | 215 | 171 | 118 |
| 1 | 202 | 130 | 201 | 125 | 250 | 089 | 071 | 240 | 173 | 212 | 162 | 175 | 156 | 164 | 114 | 192 |
| 2 | 183 | 253 | 147 | 038 | 054 | 063 | 247 | 204 | 052 | 165 | 229 | 241 | 113 | 216 | 049 | 021 |
| 3 | 004 | 199 | 035 | 195 | 024 | 150 | 005 | 154 | 007 | 018 | 128 | 226 | 235 | 039 | 178 | 117 |
| 4 | 009 | 131 | 044 | 026 | 027 | 110 | 090 | 160 | 082 | 059 | 214 | 179 | 041 | 227 | 047 | 132 |
| 5 | 083 | 209 | 000 | 237 | 032 | 252 | 177 | 091 | 106 | 203 | 190 | 057 | 074 | 076 | 088 | 207 |
| 6 | 208 | 239 | 170 | 251 | 067 | 077 | 051 | 133 | 069 | 249 | 002 | 127 | 080 | 060 | 159 | 168 |
| 7 | 081 | 163 | 064 | 143 | 146 | 157 | 056 | 245 | 188 | 182 | 218 | 033 | 016 | 255 | 243 | 210 |
| 8 | 205 | 012 | 019 | 236 | 095 | 151 | 068 | 023 | 196 | 167 | 126 | 061 | 100 | 093 | 025 | 115 |
| 9 | 096 | 129 | 079 | 220 | 034 | 042 | 144 | 136 | 070 | 238 | 184 | 020 | 222 | 094 | 011 | 219 |
| A | 224 | 050 | 058 | 010 | 073 | 006 | 036 | 092 | 194 | 211 | 172 | 098 | 145 | 149 | 228 | 121 |
| B | 231 | 200 | 055 | 109 | 141 | 213 | 078 | 169 | 108 | 086 | 244 | 234 | 101 | 122 | 174 | 008 |
| C | 186 | 120 | 037 | 046 | 028 | 166 | 180 | 198 | 232 | 221 | 116 | 031 | 075 | 189 | 139 | 138 |
| D | 112 | 062 | 181 | 102 | 072 | 003 | 246 | 014 | 097 | 053 | 087 | 185 | 134 | 193 | 029 | 158 |
| E | 225 | 248 | 152 | 017 | 105 | 217 | 142 | 148 | 155 | 030 | 135 | 233 | 206 | 085 | 040 | 223 |
| F | 140 | 161 | 137 | 013 | 191 | 230 | 066 | 104 | 065 | 153 | 045 | 015 | 176 | 084 | 187 | 022 |

可以看出，这是一种字节交织的操作，只是以一种公开的方式对字节进行移位，而和密钥没有任何关系。

也应该注意到，字节内各个比特并没有改变，因此这是一种字节顺序的操作。

行位移本质上是一组循环左移操作，左移距离分别为0、1、2和3。

$$\begin{pmatrix} c_{0,0} & c_{0,1} & c_{0,2} & c_{0,3} \\ c_{1,0} & c_{1,1} & c_{1,2} & c_{1,3} \\ c_{2,0} & c_{2,1} & c_{2,2} & c_{2,3} \\ c_{3,0} & c_{3,1} & c_{3,2} & c_{3,3} \end{pmatrix} = \begin{pmatrix} b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ b_{1,1} & b_{1,2} & b_{1,3} & b_{1,0} \\ b_{2,2} & b_{2,3} & b_{2,0} & b_{2,1} \\ b_{3,3} & b_{3,0} & b_{3,1} & b_{3,2} \end{pmatrix} \tag{9-4}$$

124

列混合是一系列特定的乘法操作：

$$\begin{pmatrix} d_{0,0} & d_{0,1} & d_{0,2} & d_{0,3} \\ d_{1,0} & d_{1,1} & d_{1,2} & d_{1,3} \\ d_{2,0} & d_{2,1} & d_{2,2} & d_{2,3} \\ d_{3,0} & d_{3,1} & d_{3,2} & d_{3,3} \end{pmatrix} = \begin{pmatrix} '02' & '03' & '01' & '01' \\ '01' & '02' & '03' & '01' \\ '01' & '01' & '02' & '03' \\ '03' & '01' & '01' & '02' \end{pmatrix} \times \begin{pmatrix} c_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ c_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} \\ c_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} \end{pmatrix} \tag{9-5}$$

式中：

'01'=00000001

'02'=00000010

'03'=00000011

所有的乘法都是在GF($2^8$)域内，列混合变换是可逆的。

每一轮最后的操作是密钥加，使用如下的方法：

$$\begin{pmatrix} e_{0,0} & e_{0,1} & e_{0,2} & e_{0,3} \\ e_{1,0} & e_{1,1} & e_{1,2} & e_{1,3} \\ e_{2,0} & e_{2,1} & e_{2,2} & e_{2,3} \\ e_{3,0} & e_{3,1} & e_{3,2} & e_{3,3} \end{pmatrix} = \begin{pmatrix} d_{0,0} & d_{0,1} & d_{0,2} & d_{0,3} \\ d_{1,0} & d_{1,1} & d_{1,2} & d_{1,3} \\ d_{2,0} & d_{2,1} & d_{2,2} & d_{2,3} \\ d_{3,0} & d_{3,1} & d_{3,2} & d_{3,3} \end{pmatrix} \oplus \begin{pmatrix} k_{0,0} & k_{0,1} & k_{0,2} & k_{0,3} \\ k_{1,0} & k_{1,1} & k_{1,2} & k_{1,3} \\ k_{2,0} & k_{2,1} & k_{2,2} & k_{2,3} \\ k_{3,0} & k_{3,1} & k_{3,2} & k_{3,3} \end{pmatrix} \tag{9-6}$$

轮密钥用以下方法产生。将原始的128比特密钥排列成4×4的字节矩阵。这一矩阵可以看成由4列组成，分别是$W(0)$、$W(1)$、$W(2)$和$W(3)$。总共40列，后面的是$W(4)$，……，$W(43)$。

第$i$轮的轮密钥由列$W(i)$、$W(i+1)$、$W(i+2)$和$W(i+3)$组成。如果$i$是4的整数倍，那么$W(i)=W(i-4)$^$T(W(i-1))$，这里的$T$是对列$W(i-1)$中$a$、$b$、$c$和$d$的变换。

- ❑ 循环移位得到$b$、$c$、$d$和$a$。
- ❑ 使用S盒替换每一个字节，得到$e$、$f$、$g$和$h$。
- ❑ 在GF($2^8$)域内计算轮常数$r(i)$=00000010$(i-4)/4$。
- ❑ $T(W(i-1))=(e \oplus r(i), f, g, h)$。

如果$i$不是4的整数倍，那么$W(i)=W(i-4) \oplus W(i-1)$。

125

## AES的VHDL实现

用VHDL实现块加密有两种选择，在这里使用结构化方法（本章前面的DES例子中有说明）。有时定义一个函数库很有意义，使用这些库函数可以完成一些简单的模型。

下面的AES例子中，定义了一个顶层实体和构造体，其中具有最小化的结构，并且完全是用函数定义的。这一点在使用行为级综合工具时显得特别有用，因为它给了结构优化最大

的灵活性：

```
library ieee;
use ieee.std_logic_1164.all;
entity AES is
        port (
                plaintext : in std_logic_vector(127 downto 0);
                keytext : in std_logic_vector(127 downto 0);
                encrypt : in std_logic;
                go : in std_logic;
                ciphertext : out std_logic_vector
                  (127 downto 0);
                done : out std_logic := '0'
        )
end;

use work.aes_functions.all;
architecture behaviour of AES is
begin
        process
        begin
                wait until go = '1';
                done <= '0';
                ciphertext <= aes_core(plaintext, keytext,
                  encrypt);
                done <= '1';
        end process;
end;
```

126

在本例中，输入明文和密钥都定义成了128比特的向量，输出密文也是128比特宽。信号go对加密过程进行初始化，信号done则表示加密过程结束。

请注意，我们使用了一个被称为aes_functions的工作库，这个库中封装了AES算法需要的所有相关函数。函数以包的形式（aes_functions）定义，具体的VHDL代码如下：

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
package aes_functions is

        constant nr : integer := 10;
        constant nb : integer := 4;
        constant nk : integer := 4;

        subtype vec1408 is std_logic_vector(1407 downto 0);
        subtype vec128 is std_logic_vector(127 downto 0);
        subtype vec64 is std_logic_vector(63 downto 0);
        subtype vec32 is std_logic_vector(31 downto 0);
        subtype vec16 is std_logic_vector(15 downto 0);
```

```
      subtype vec8 is std_logic_vector(7 downto 0);

subtype int9 is integer range 0 to 9;

      function input_output (input : vec128) return
        vec128;
      function sBox (pt : vec8)      return vec8;
      function subBytes (plaintext : vec128) return
        vec128;
      function shiftRows (plaintext : vec128) return
        vec128;
      function ffmul(pt : vec8; mul : vec8) return vec8;
      function mixCL( l0 : vec8; l1 : vec8; l2 : vec8;
        l3 : vec8) return vec8;
      function mixColumns(pt : vec128) return vec128;
      function rcon (input : int9) return vec8;
      function aes_keyexpansion(key : vec128) return
        vec1408;
      function aes_core (signal plaintext : vec128;
        signal keytext : vec128; signal encrypt :
        std_logic) return vec128;
end;

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
package body aes_functions is
--------------------------------------------------------
----------------------------
    function subBytes (plaintext : vec128)
--  moods inline
    return vec128 is
         variable ciphertext : vec128;
    begin
    ciphertext := sBox(plaintext(127 DOWNTO 120)) &
                  sBox(plaintext(119 DOWNTO 112)) &
                  sBox(plaintext(111 DOWNTO 104)) &
                  sBox(plaintext(103 DOWNTO 96)) &
                  sBox(plaintext(95 DOWNTO 88)) &
                  sBox(plaintext(87 DOWNTO 80)) &
                  sBox(plaintext(79 DOWNTO 72)) &
                  sBox(plaintext(71 DOWNTO 64)) &
                  sBox(plaintext(63 DOWNTO 56)) &
                  sBox(plaintext(55 DOWNTO 48)) &
                  sBox(plaintext(47 DOWNTO 40)) &
                  sBox(plaintext(39 DOWNTO 32)) &
```

127

```
                    sBox(plaintext(31 DOWNTO 24)) &
                    sBox(plaintext(23 DOWNTO 16)) &
                    sBox(plaintext(15 DOWNTO 8)) &
                    sBox(plaintext(7 DOWNTO 0));
    return ciphertext;
    end;

------------------------------------------------------------
----------------------------
    function shiftRows (plaintext : vec128)
--  moods inline
    return vec128 is
         variable ciphertext : vec128;
    begin
         -- line 0 (the first): no shift
    ciphertext := plaintext(31 DOWNTO 24) &
                  plaintext(55 DOWNTO 48) &
                  plaintext(79 DOWNTO 72) &
                  plaintext(103 DOWNTO 96) &
                  plaintext(127 DOWNTO 120) &
                  plaintext(23 DOWNTO 16) &
                  plaintext(47 DOWNTO 40) &
                  plaintext(71 DOWNTO 64) &
                  plaintext(95 DOWNTO 88) &
                  plaintext(119 DOWNTO 112) &
                  plaintext(15 DOWNTO 8) &
                  plaintext(39 DOWNTO 32) &
                  plaintext(63 DOWNTO 56) &
                  plaintext(87 DOWNTO 80) &
                  plaintext(111 DOWNTO 104) &
                  plaintext(7 DOWNTO 0);
    return ciphertext;
    end;
------------------------------------------------------------
----------------------------
    function tableLog (input : vec8)
--  moods inline
    return vec8 is
           variable output : vec8;
           type table256 is array(0 to 255) of vec8;
           constant pt_256 : table256 := (
           --moods rom
                  X"00",  X"00",  X"19",   X "01",  X "32",
                  X"02",   X"1a",   X "c6",   X"4b",
                  X"c7",   X"1b",   X "68",   X"33",
                  X"ee",  X"df",  X"03",   X"64",
```

```
X"04",    X"e0",    X"0e",    X"34",
X"8d",    X"81",    X"ef",    X"4c",
X"71",    X"08",    X"c8",    X"f8",
X"69",    X"1c",    X"c1",    X"7d",

X"c2",    X"1d",    X"b5",    X"f9",
X"b9",    X"27",    X"6a",    X"4d",
X"e4",    X"a6",    X"72",    X"9a",
X"c9",    X"09",    X"78",    X"65",

X"2f",    X"8a",    X"05",    X"21",
X"0f",    X"e1",    X"24",    X"12",
X"f0",    X"82",    X"45",    X"35",
X"93",    X"da",    X"8e",    X"96",

X"8f",    X"db",    X"bd",    X"36",
X"d0",    X"ce",    X"94",    X"13",
X"5c",    X"d2",    X"f1",    X"40",
X"46",    X"83",    X"38",    X"66",

X"dd",    X"fd",    X"30",    X"bf",
X"06",    X"8b",    X"62",    X"b3",
X"25",    X"e2",    X"98",    X"22",
X"88",    X"91",    X"10",    X"7e",

X"6e",    X"48",    X"c3",    X"a3",
X"b6",    X"1e",    X"42",    X"3a",
X"6b",    X"28",    X"54",    X"fa",
X"85",    X"3d",    X"ba",    X"2b",

X"79",    X"0a",    X"15",    X"9b",
X"9f",    X"5e",    X"ca",    X"4e",
X"d4",    X"ac",    X"e5",    X"f3",
X"73",    X"a7",    X"57",    X"af",

X"58",    X"a8",    X"50",    X"f4",
X"ea",    X"d6",    X"74",    X"4f",
X"ae",    X"e9",    X"d5",    X"e7",
X"e6",    X"ad",    X"e8",    X"2c",

X"d7",    X"75",    X"7a",    X"eb",
X"16",    X"0b",    X"f5",    X"59",
X"cb",    X"5f",    X"b0",    X"9c",
X"a9",    X"51",    X"a0",    X"7f",

X"0c",    X"f6",    X"6f",    X"17",
X"c4",    X"49",    X"ec",    X"d8",
X"43",    X"1f",    X"2d",    X"a4",
X"76",    X"7b",    X"b7",    X"cc",
```

```
                        X"bb",    X"3e",    X"5a",    X"fb",
                        X"60",    X"b1",    X"86",    X"3b",
                        X"52",    X"a1",    X"6c",    X"aa",
                        X"55",    X"29",    X"9d",    X"97",

                        X"b2",    X"87",    X"90",    X"61",
                        X"be",    X"dc",    X"fc",    X"bc",
                        X"95",    X"cf",    X"cd",    X"37",
                        X"3f",    X"5b",    X"d1",    X"53",

                        X"39",    X"84",    X"3c",    X"41",
                        X"a2",    X"6d",    X"47",    X"14",
                        X"2a",    X"9e",    X"5d",    X"56",
                        X"f2",    X"d3",    X"ab",    X"44",

                        X"11",    X"92",    X"d9",    X"23",
                        X"20",    X"2e",    X"89",    X"b4",
                        X"7c",    X"b8",    X"26",    X"77",
                        X"99",    X"e3",    X"a5",    X"67",

                        X"4a",    X"ed",    X"de",    X"c5",
                        X"31",    X"fe",    X"18",    X"0d",
                        X"63",    X"8c",    X"80",    X"c0",
                        X"f7",    X"70",    X"07" );
begin
        output := pt_256(TO_INTEGER(UNSIGNED(input)));
        return output;
end;

--------------------------------------------------------
-----------------------------
    function tableExp (input : vec8)
--  moods inline
    return vec8 is
          variable output : vec8;
          type table256 is array(0 to 255) of vec8;
          constant pt_256 : table256 := (
          --moods rom
                        X"01",    X"03",    X"05",    X"0f",
                        X"11",    X"33",    X"55",    X"ff",
                        X"1a",    X"2e",    X"72",    X"96",
                        X"a1",    X"f8",    X"13",    X"35",

                        X"5f",    X"e1",    X"38",    X"48",
                        X"d8",    X"73",    X"95",    X"a4",
                        X"f7",    X"02",    X"06",    X"0a",
                        X"1e",    X"22",    X"66",    X"aa",
```

```
X"e5",   X"34",   X"5c",   X"e4",
X"37",   X"59",   X"eb",   X"26",
X"6a",   X"be",   X"d9",   X"70",
X"90",   X"ab",   X"e6",   X"31",

X"53",   X"f5",   X"04",   X"0c",
X"14",   X"3c",   X"44",   X"cc",
X"4f",   X"d1",   X"68",   X"b8",
X"d3",   X"6e",   X"b2",   X"cd",

X"4c",   X"d4",   X"67",   X"a9",
X"e0",   X"3b",   X"4d",   X"d7",
X"62",   X"a6",   X"f1",   X"08",
X"18",   X"28",   X"78",   X"88",

X"83",   X"9e",   X"b9",   X"d0",
X"6b",   X"bd",   X"dc",   X"7f",
X"81",   X"98",   X"b3",   X"ce",
X"49",   X"db",   X"76",   X"9a",

X"b5",   X"c4",   X"57",   X"f9",
X"10",   X"30",   X"50",   X"f0",
X"0b",   X"1d",   X"27",   X"69",
X"bb",   X"d6",   X"61",   X"a3",

X"fe",   X"19",   X"2b",   X"7d",
X"87",   X"92",   X"ad",   X"ec",
X"2f",   X"71",   X"93",   X"ae",
X"e9",   X"20",   X"60",   X"a0",

X"fb",   X"16",   X"3a",   X"4e",
X"d2",   X"6d",   X"b7",   X"c2",
X"5d",   X"e7",   X"32",   X"56",
X"fa",   X"15",   X"3f",   X"41",

X"c3",   X"5e",   X"e2",   X"3d",
X"47",   X"c9",   X"40",   X"c0",
X"5b",   X"ed",   X"2c",   X"74",
X"9c",   X"bf",   X"da",   X"75",

X"9f",   X"ba",   X"d5",   X"64",
X"ac",   X"ef",   X"2a",   X"7e",
X"82",   X"9d",   X"bc",   X"df",
X"7a",   X"8e",   X"89",   X"80",

X"9b",   X"b6",   X"c1",   X"58",
X"e8",   X"23",   X"65",   X"af",
```

```
                        X"ea",    X"25",    X"6f",    X"b1",
                        X"c8",    X"43",    X"c5",    X"54",

                        X"fc",    X"1f",    X"21",    X"63",
                        X"a5",    X"f4",    X"07",    X"09",
                        X"1b",    X"2d",    X"77",    X"99",
                        X"b0",    X"cb",    X"46",    X"ca",

                        X"45",    X"cf",    X"4a",    X"de",
                        X"79",    X"8b",    X"86",    X"91",
                        X"a8",    X"e3",    X"3e",    X"42",
                        X"c6",    X"51",    X"f3",    X"0e",

                        X"12",    X"36",    X"5a",    X"ee",
                        X"29",    X"7b",    X"8d",    X"8c",
                        X"8f",    X"8a",    X"85",    X"94",
                        X"a7",    X"f2",    X"0d",    X"17",

                        X"39",    X"4b",    X"dd",    X"7c",
                        X"84",    X"97",    X"a2",    X"fd",
                        X"1c",    X"24",    X"6c",    X"b4",
                        X"c7",    X"52",    X"f6",    X"01");
    begin
           output := pt_256(TO_INTEGER(UNSIGNED(input)));
    return output;
    end;
--------------------------------------------------------------
------------------------
    function ffmul(pt : vec8; mul : vec8)
--  moods inline
    return vec8 is
              --variable res : vec8;
              variable tablogpt : vec8;
              variable tablogmul : vec8;
              variable tablogpt8 : unsigned(8 downto 0);
              variable tablogmul8 : unsigned(8 downto 0);
              variable carrie : std_logic_vector (8 downto 0);
              variable power : vec8;
    variable result : vec8;
      begin
    tablogpt := tableLog(pt);
    tablogmul := tableLog(mul);

    tablogpt8 := unsigned("0" & tablogpt);
    tablogmul8 := unsigned("0" & tablogmul);

    carrie := std_logic_vector(tablogmul8 + tablogpt8);
```

```
    if pt = X"00" or mul = X"00" then
              result := X"00";
      elsif carrie(8) = '1' or carrie(7 DOWNTO 0) =
        X"ff" then -- mod 255
          power := std_logic_vector(unsigned(carrie
            (7 DOWNTO 0)) + 1 ); -- power = power - 255
          result := tableExp(power);
    else
          power := carrie(7 DOWNTO 0);
    result := tableExp(power);
    end if;
  return result;
    end;
-----------------------------------------------------------
-------------------------
  function mixCL(   l0 : vec8; l1 : vec8; l2 : vec8;
    l3 : vec8)
--  moods inline
  return vec8 is
       variable ct : vec8;
  begin
         ct := ffmul(l0, X"02")xor ffmul(l1, X"01")
           xor ffmul(l2, X"01")xor ffmul(l3, X"03");
         return ct;
  end;
-----------------------------------------------------------
-------------------------
  function mixColumns(pt : vec128)
--  moods inline
  return vec128 is
       variable ct : vec128;
  begin
         ct := mixCL(pt(127 DOWNTO 120), pt(119 DOWNTO
           112), pt(111 DOWNTO 104), pt(103 DOWNTO 96)) &
         mixCL(pt(119 DOWNTO 112), pt(111 DOWNTO 104),
           pt(103 DOWNTO 96), pt(127 DOWNTO 120)) &
         mixCL(pt(111 DOWNTO 104), pt(103 DOWNTO 96),
           pt(127 DOWNTO 120), pt(119 DOWNTO 112)) &
         mixCL(pt(103 DOWNTO 96), pt(127 DOWNTO 120),
           pt(119 DOWNTO 112), pt(111 DOWNTO 104)) &

         mixCL(pt(95 DOWNTO 88), pt(87 DOWNTO 80),
           pt(79 DOWNTO 72), pt(71 DOWNTO 64)) &
         mixCL(pt(87 DOWNTO 80), pt(79 DOWNTO 72), pt(71
           DOWNTO 64), pt(95 DOWNTO 88)) &
         mixCL(pt(79 DOWNTO 72),pt(71 DOWNTO 64), pt(95
```

133

```
          DOWNTO 88), pt(87 DOWNTO 80)) &
        mixCL(pt(71 DOWNTO 64), pt(95 DOWNTO 88), pt(87
          DOWNTO 80), pt(79 DOWNTO 72)) &
        mixCL(pt(63 DOWNTO 56), pt(55 DOWNTO 48), pt(47
          DOWNTO 40), pt(39 DOWNTO 32)) &
        mixCL(pt(55 DOWNTO 48), pt(47 DOWNTO 40), pt(39
          DOWNTO 32), pt(63 DOWNTO 56)) &
        mixCL(pt(47 DOWNTO 40), pt(39 DOWNTO 32), pt(63
          DOWNTO 56), pt(55 DOWNTO 48)) &
        mixCL(pt(39 DOWNTO 32), pt(63 DOWNTO 56), pt(55
          DOWNTO 48), pt(47 DOWNTO 40)) &

       mixCL(pt(31 DOWNTO 24), pt(23 DOWNTO 16), pt(15
         DOWNTO 8), pt(7 DOWNTO 0)) &
        mixCL(pt(23 DOWNTO 16), pt(15 DOWNTO 8), pt(7
          DOWNTO 0), pt(31 DOWNTO 24)) &
        mixCL(pt(15 DOWNTO 8), pt(7 DOWNTO 0), pt(31
          DOWNTO 24), pt(23 DOWNTO 16)) &
        mixCL(pt(7 DOWNTO 0), pt(31 DOWNTO 24), pt(23
          DOWNTO 16), pt(15 DOWNTO 8));
        return ct;
    end;
---------------------------------------------------------
----------------------------
    function input_output (input : vec128)
--  moods inline
    return vec128 is
         variable output : vec128;
    function flip(input : vec32) return vec32 is
    -- moods inline
    begin
      return input(7 DOWNTO 0) & input(15 DOWNTO 8) &
        input(23 DOWNTO 16) & input(31 DOWNTO 24);
    end;

    begin
           return flip(input(127 downto 96)) & flip(input
             (95 downto 64)) & flip(input(63 downto 32)) &
             flip(input(31 downto 0));
    end;
---------------------------------------------------------
----------------------------
    function aes_keyexpansion(key : vec128)
--  moods inline
    return vec1408 is
         variable iok : vec128;
```

```
      variable er0,er1,er2,er3,er4,er5,er6,er7,
        er8, er9 : vec128;
    -- variable zero : vec128;
    -- variable expandedkeys : vec1408;

  function exp_round(input : vec128; round : int9)
  return vec128 is
--  moods inline
     variable r1,r2,r3,r4,r5 : vec32;
  begin
        r1 := sBox(input(7 downto 0)) &
              sBox(input(31 downto 24)) &
              sBox(input(23 downto 16)) &
              (sBox(input(15 downto 8)) xor rcon(round));

        r2 := input(127 downto 96)xor r1;
        r3 := input(95 downto 64)xor r2;
        r4 := input(63 downto 32)xor r3;
        r5 := input(31 downto 0)xor r4;
        return r2 & r3 & r4 & r5;
  end;

  begin
    -- First Round
    iok := input_output(key);
    -- other rounds
    er9 := exp_round(iok,9);
    er8 := exp_round(er9,8);
    er7 := exp_round(er8,7);
    er6 := exp_round(er7,6);
    er5 := exp_round(er6,5);
    er4 := exp_round(er5,4);
    er3 := exp_round(er4,3);
    er2 := exp_round(er3,2);
    er1 := exp_round(er2,1);
    er0 := exp_round(er1,0);

    return (iok & er9 & er8 & er7 & er6 & er5 & er4 &
      er3 & er2 & er1 & er0);
  end;
------------------------------------------------------------
---------------------------
  function aes_core (signal plaintext : vec128; signal
  keytext : vec128; signal encrypt : std_logic)
--  moods inline
  return vec128 is
```

134

```
        variable rk0 : vec128;
        variable ciphertext, expkey : vec128;
        variable ct1,ct2,ct3,ct4,ct5,ct6,ct7,ct8 : vec128;
        variable expandedkeys : vec1408;
    begin
          -- expanded key schedule
          expandedkeys := aes_keyexpansion(keytext);

          -- Round 0
          ct1 := input_output(plaintext) xor
            expandedkeys(1407 downto 1280);
    -- Round 1 to Nr-1
    -- for i in 1 to Nr-1 loop
    for i in 1 to 9 loop
                 ct2 := subBytes(ct1);
                 ct3 := shiftRows(ct2);
                 ct4 := mixColumns(ct3);
      case(i)is
        when 1 => expkey := expandedkeys(1279 downto 1152);
        when 2 => expkey := expandedkeys(1151 downto 1024);
        when 3 => expkey := expandedkeys(1023 downto 896);
        when 4 => expkey := expandedkeys(895 downto 768);
        when 5 => expkey := expandedkeys(767 downto 640);
        when 6 => expkey := expandedkeys(639 downto 512);
        when 7 => expkey := expandedkeys(511 downto 384);
        when 8 => expkey := expandedkeys(383 downto 256);
        when 9 => expkey := expandedkeys(255 downto 128);
        when others => null;
      end case;
                 ct1 := ct4 xor expkey;
            end loop;

            -- Final Round Nr=10
            ct5 := subBytes(ct1);
            ct6 := shiftRows(ct5);
            ct7 := ct6 xor expandedkeys(1407-128*Nr downto
              1280-128*Nr);
            ciphertext := input_output(ct7);
            return ciphertext;
    end;
--------------------------------------------------------------
--------------------------
    function rcon (input : int9)
--  moods inline
    return vec8 is
    type rcont_t is array(0 to 9) of vec8;
```

[135]

```
    constant table_rcon : rcont_t := (
    -- moods rom
    X"36", X"1b", X"80", X"40", X"20", X"10", X"08", X"04",
       X"02", X"01");
    begin
             return table_rcon(input);
    end;
------------------------------------------------------------
----------------------------
    function sBox (pt : vec8)
--  moods inline
    return vec8 is
         variable ct : vec8;
         type table256 is array(0 to 255) of vec8;
         constant pt_256 : table256 := (
         -- moods rom
X"63",  X"7c",  X"77",  X"7b",  X"f2",  X"6b",  X"6f",  X"c5",
X"30",  X"01",  X"67",  X"2b",  X"fe",  X"d7",  X"ab",  X"76",
X"ca",  X"82",  X"c9",  X"7d",  X"fa",  X"59",  X"47",  X"f0",
X"ad",  X"d4",  X"a2",  X"af",  X"9c",  X"a4",  X"72",  X"c0",
X"b7",  X"fd",  X"93",  X"26",  X"36",  X"3f",  X"f7",  X"cc",
X"34",  X"a5",  X"e5",  X"f1",  X"71",  X"d8",  X"31",  X"15",
X"04",  X"c7",  X"23",  X"c3",  X"18",  X"96",  X"05",  X"9a",
X"07",  X"12",  X"80",  X"e2",  X"eb",  X"27",  X"b2",  X"75",
X"09",  X"83",  X"2c",  X"1a",  X"1b",  X"6e",  X"5a",  X"a0",
X"52",  X"3b",  X"d6",  X"b3",  X"29",  X"e3",  X"2f",  X"84",
X"53",  X"d1",  X"00",  X"ed",  X"20",  X"fc",  X"b1",  X"5b",
X"6a",  X"cb",  X"be",  X"39",  X"4a",  X"4c",  X"58",  X"cf",
X"d0",  X"ef",  X"aa",  X"fb",  X"43",  X"4d",  X"33",  X"85",
X"45",  X"f9",  X"02",  X"7f",  X"50",  X"3c",  X"9f",  X"a8",
X"51",  X"a3",  X"40",  X"8f",  X"92",  X"9d",  X"38",  X"f5",
X"bc",  X"b6",  X"da",  X"21",  X"10",  X"ff",  X"f3",  X"d2",
X"cd",  X"0c",  X"13",  X"ec",  X"5f",  X"97",  X"44",  X"17",
X"c4",  X"a7",  X"7e",  X"3d",  X"64",  X"5d",  X"19",  X"73",
X"60",  X"81",  X"4f",  X"dc",  X"22",  X"2a",  X"90",  X"88",
X"46",  X"ee",  X"b8",  X"14",  X"de",  X"5e",  X"0b",  X"db",
X"e0",  X"32",  X"3a",  X"0a",  X"49",  X"06",  X"24",  X"5c",
X"c2",  X"d3",  X"ac",  X"62",  X"91",  X"95",  X"e4",  X"79",
X"e7",  X"c8",  X"37",  X"6d",  X"8d",  X"d5",  X"4e",  X"a9",
X"6c",  X"56",  X"f4",  X"ea",  X"65",  X"7a",  X"ae",  X"08",
X"ba",  X"78",  X"25",  X"2e",  X"1c",  X"a6",  X"b4",  X"c6",
X"e8",  X"dd",  X"74",  X"1f",  X"4b",  X"bd",  X"8b",  X"8a",
X"70",  X"3e",  X"b5",  X"66",  X"48",  X"03",  X"f6",  X"0e",
X"61",  X"35",  X"57",  X"b9",  X"86",  X"c1",  X"1d",  X"9e",
X"e1",  X"f8",  X"98",  X"11",  X"69",  X"d9",  X"8e",  X"94",
X"9b",  X"1e",  X"87",  X"e9",  X"ce",  X"55",  X"28",  X"df",
```

136

```
X"8c",  X"a1",  X"89",  X"0d",  X"bf",  X"e6",  X"42",  X"68",
X"41",  X"99",  X"2d",  X"0f",  X"b0",  X"54",  X"bb",  X"16");
begin
        ct := pt_256(TO_INTEGER(UNSIGNED(pt)));
        return ct;
end;

--------------------------------------------------------
end;
```

定义完函数之后，再定义顶层实体，然后可以定义一个测试平台，对AES模块施加一系列输入数据，并验证其输出是否正确。需要注意的是，我们在测试平台中使用了断言（assertion）技术，以确定整个操作的正确性：

```
library ieee;
use ieee.std_logic_1164.all;
entity testAES is
end;

library ieee;
use ieee.std_logic_1164.all;
use work.aes_functions.all;
architecture behaviour of testAES is

      component aes
            port(
                 plaintext : in std_logic_vector (127
                   downto 0);
                 keytext : in std_logic_vector(127
                   downto 0);
                 encrypt : in std_logic;
                 go : in std_logic;
                 ciphertext : out std_logic_vector
                   (127 downto 0);
                 done : out std_logic
            );
      end component;

      for all : aes use entity work.aes;

      signal plaintext : std_logic_vector(127 downto 0);
      signal keytext : std_logic_vector(127 downto 0);
      signal encrypt : std_logic;
      signal go : std_logic := '0';
      signal ciphertext : std_logic_vector(127 downto 0);
      signal done : std_logic;
      signal ok : std_logic := '0';
```

137

```
begin
      plaintext <= X"00000000000000000000000000000000",
        X"3243f6a8885a308d313198a2e0370734" after 50 ns ;
      keytext <= X"00000000000000000000000000000000",
        X"2b7e151628aed2a6abf7158809cf4f3c" after 100 ns;

      process (ciphertext)
                    variable ct : std_logic_vector(127
                      downto 0) :=
                      X"3925841d02dc09fbdc118597196a0b32";
      begin
             assert ct = ciphertext
                          report "Test vectors do not match"
                          severity note;
                      assert not (ct = ciphertext)
                               report "Test vectors Matched"
                               severity note;
      end process;

      process
      begin
                      go <= not go after 20 ns;
      end process;

      DUT : aes port map (plaintext, keytext, encrypt,
        go, ciphertext, done);
end;
```

138

## 9.5 小结

本章举例说明了如何用VHDL实现两种标准的块加密器：DES和AES。这两种算法现在被广泛使用，而且大量应用于各种硬件中。还有许多其他的方法，因为安全领域需要加密技术不断进步。毫无疑问，在将来，更安全、更健壮的加密算法仍将出现，也需要用VHDL实现。

139

# 第10章 存 储 器

## 10.1 引言

我们先以同步动态随机存取存储器（SDRAM，Synchronous Dynamic Random Access Memory）为例进行说明，这种类型存储器的主要特征列举如下。

- DRAM依靠逻辑门上的晶体管电容来存储数据。
- DRAM比SRAM（静态随机存取存储器）更小，更紧凑。
- DRAM无法进行综合，必须使用一个单独的DRAM芯片。
- SDRAM需要一个同步时钟，这个时钟要与硬件系统（同微处理器一同进行操作）其他部分的时钟具有一致性。
- DRAM中保存的数据必须定时刷新，因为存储的电荷有衰减现象。
- DRAM的速度要慢于SRAM。

静态RAM（SRAM）的工作方式与只读存储器（ROM）类似，也具有一些重要的特征需要注意。

- 存储器的存储单元基于标准的锁存器。
- SRAM速度更快。
- SRAM比DRAM（或者SDRAM）更大。
- SRAM可以被综合到FPGA内，所以对于小规模、快速的寄存器或存储器模块非常理想。



静态RAM本质上是异步的，但是也可以修改为同步的（就像SDRAM就是DRAM的同步版本一样），并且常被称为同步RAM。

从这一点来看，Flash存储器是有用的，即使它的操作与前面介绍的存储器完全不同，因为Flash存储器使用起来非常容易，而且在FPGA开发板上普遍使用。

Flash存储器本质上是一种EEPROM（电可编程只读存储器），可以作为一种耐久性的存储器使用。为什么是耐久性的呢？因为在Flash存储器中，即使断掉电源，存储器中存储的数据也不会丢失，所以常被作为一种只读存储器来使用。在FPGA系统中，Flash存储器常被用来存储FPGA程序，不过也可以用作随机存储器存储当前的数据。

## 10.2 用VHDL对存储器进行建模

用VHDL对存储器进行建模时必须非常小心。由于某些存储器不能综合，如果使用模型的话，必须反映片外真实器件的正确的物理行为。这尤其适用于存取时间和时序违规条件。如果违反了时序要求，那么数据在最好的情况下也不可信，在最差的情况下则完全无用。设计人员可能会发现他们处于一种非常尴尬的境地，因为仿真模型工作得很好，但实际的硬件却完全无法工作。

本章所有的模型中，都使用不带物理延迟的VHDL描述，但是如果这些模型要用于实际

的系统中，这些延迟特性必须增加进去。

## 10.3 只读存储器

只读存储器（ROM）从本质上看就是一组存储在寄存器中的预定义的数据。存储器有两个定义，一个是存储区的数量，另一个是位宽。例如，如果有16个存储位置，每一个存储位置为8比特，那么这个存储器就是16x8的ROM。基本的ROM有一个输入，用来定义要访问的地址，还有一个输出，用来放置被访问地址处的数据。以下是用VHDL描述的ROM行为模型的实体：

141

```
ENTITY ROM16x8 IS
  PORT (address : IN INTEGER RANGE 0 TO 15;
       dout : OUT std_logic_vector (7 DOWNTO 0));
END ENTITY ROM16x8;
```

可以看出，地址被定义为整数类型，但是范围限制在了ROM的地址空间内。ROM的构造体定义成了一个固定的数组，可以直接访问。下面给出的ROM例子中使用了一组数据，代码如下：

```
ARCHITECTURE example OF rom16x8 IS
    TYPE romdata IS ARRAY (0 TO 15)
    OF std_logic_vector( 7 DOWNTO 0);
CONSTANT romvals : romdata := ("00000000",
                               "01010011",
                               "01110010",
                               "01101100",
                               "01110101",
                               "11010111",
                               "11011111",
                               "00111110",
                               "11101100",
                               "10000110",
                               "11111001",
                               "00111001",
                               "01010101",
                               "11110111",
                               "10111111",
                               "11101101");
BEGIN
  data <= romvals(address);
END ARCHITECTURE example;
```

如果要在其他例子中使用这个ROM，首先需要在VHDL的测试平台中声明ROM，然后使用整数变量作为地址。下面给出一个测试平台的例子。

```
library ieee;
use ieee.std_logic_1164.all;

entity testrom is
end entity testrom;

architecture test of testrom is
```

```
    signal address : integer := 0;
    signal data : std_logic_vector ( 7 downto 0 );
begin
    rom16x8 : entity work.rom16x8 (example)
        port map ( address, data );
end architecture test;
```

[142]

需要注意的是，std_logic_vector 类型的数据需要使用IEEE库std_logic_1164，信号数据的值依赖于给出的地址。

## 10.4 随机存取存储器

DRAM是一种二维的存储器，具有格状结构，分别用行地址和列地址访问。DRAM是异步的，没有时钟。这就意味着在访问DRAM时必须注意相应的时序，以保证整个传输过程的数据完整性。

以下的VHDL模型有一个地址输入和两个控制信号RADDR和CADDR（分别用来指定行地址和列地址）。还有一个RW信号定义读写，高电平时表示读，低电平时表示写。最后，数据放在INOUT类型（双向）信号DATA中。最终的VHDL实体如下列代码所示。在这个例子中，行数为$2^8$，列数也是$2^8$。所以总的数据存储量为1Mbit。

```
ENTITY DRAM1MB IS
 PORT (
      address : IN INTEGER RANGE 0 TO 2**8-1;
      RW : std_logic;
      data : OUT std_logic_vector (15 DOWNTO 0));
END ENTITY DRAM1MB;
```

VHDL构造体如下：

```
architecture behav of DRAM1MB is
begin
process (RADDR, CADDR, RW) is
      type dram is array (0 to 2**16 - 1) of
                      std_logic_vector(15 downto 0);
  variable radd : INTEGER range 0 to 2**8 - 1;
  variable madd : INTEGER range 0 to 2**16 - 1;
  variable memory : dram;
begin
  data <= (others => 'Z');
  if falling_edge(RADDR) then
    radd := address;
  elsif falling_edge(CADDR) then
    madd := radd*2**18 +Address;
    if RADDR = '0' and RW = '0' then
      memory(madd) := data;
    end if;
  elsif CADDR = '0' and RADDR = '0' and RW = '1' then
    data <= memory(madd);
    end if;
end process;
```

[143]

```
end architecture behav;
```

在测试平台中利用这个模型就可以将数据写入某个地址，然后再将另一个数据写入另一个地址，然后再读出原来地址的数据。这个测试平台的VHDL代码如下：

```
library ieee;
use ieee.std_logic_1164.all;

entity testram is
end entity testram;

architecture test of testram is
      signal address : integer range 0 to 2**8-1 := 0;
      signal rw : std_logic;
      signal c : std_logic;
      signal r : std_logic;
      signal data : std_logic_vector ( 15 downto 0 );
begin
      dram: entity work.dram1mb(behav)
            port map ( address, rw, c, r, data );
      address <= 23 after 0 ns, 47 after 30 ns, 23 after
        90 ns;
      rw <=   '0' after 0 ns, '1' after 90 ns;
      c <=    '1' after 0 ns, '0' after 20 ns,
              '1' after 50 ns, '0' after 70 ns,
              '1' after 90 ns, '0' after 100 ns;
      r <=    '1' after 0 ns, '0' after 10 ns,
              '1' after 40 ns, '0' after 60 ns,
              '1' after 80 ns, '0' after 100 ns;
      data <= X"1234" after 0 ns, X"5678" after 40 ns;
end architecture test;
```

对这个模型的测试结果可以从波形中看出来（如图10-1所示），其中显示了地址、数据

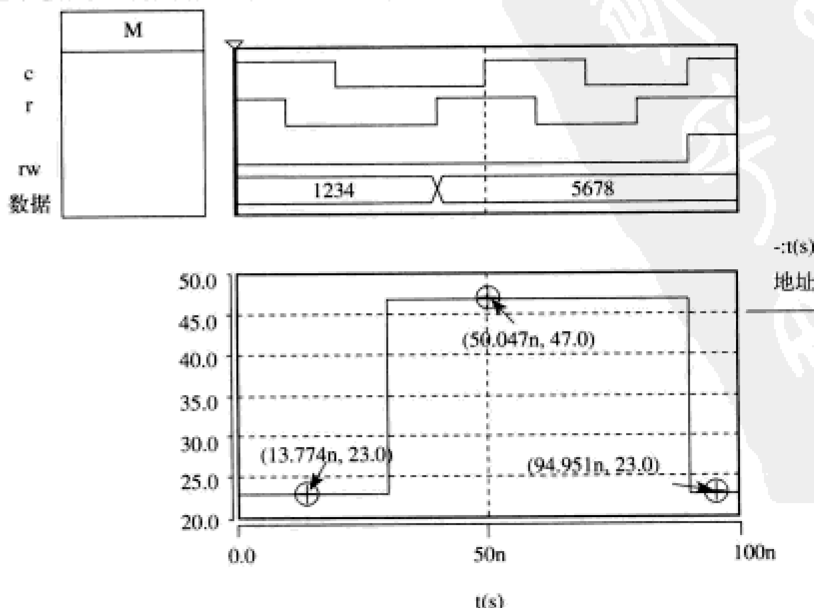



图10-1 DRAM仿真结果

和控制信号的正确行为。

这里有一点需要注意，这个RAM模型并没有任何实际电路中真实存在的延迟，如果这些延迟对设计的功能非常重要，那么必须将它们增加到模型中。

144

## 10.5 SRAM

前文介绍了如何访问异步存储器，这里再讨论一下SRAM（需要一个时钟信号）。大多数实际的设计中，RAM一般都用独立的存储器芯片在片外实现。但有时候为了快速访问，需要在FPGA内实现一个很小的RAM，或者在需要频繁访问小块RAM的硬件周围实现一个局部存储器。

一般来说，存储器的设计约束比其他功能模块更加严格，因为用触发器存储数据时不需要用太多查找表（LUT）中的其他逻辑，但是面积比较大。还以FPGA设计为例，权衡的标准是，使用内部RAM带来的速度和性能上的提升超过了由此增加的面积。

145

从设计的角度看，SRAM的VHDL模型非常类似于前面介绍的基本异步RAM模型。唯一的区别是SRAM在有了地址信号后不是立即（或很短一段延迟后）输出数据，SRAM的数据只能在时钟沿（是上升沿还是下降沿取决于设计的需求）存取。

考察一下SRAM的VHDL实体就会发现，存储器大小为$2^m$，数据总线宽度为$n$，实体如下所示。这个VHDL模型有两个参数$m$和$n$。默认情况下，$m$为10，可提供1024个地址空间，总线宽度$n$为8，因此这个RAM总共有8kbit存储空间。很明显，任意大小和宽度的RAM都可以实现，根据计算类型可以得到特定的存储器模块。

```
ENTITY SRAM IS
  GENERIC (
    M : natural := 10;
    N : natural := 8
    );
  PORT (
    clk : in std_logic;
    addr : in std_logic_vector(m-1 downto 0);
    wr : in std_logic;
    d : in std_logic_vector (n-1 downto 0);
    q : out std_logic_vector (n-1 downto 0)
    );
END ENTITY DRAM1MB;
```

注意，这里有两个控制信号：时钟信号clk和写使能信号wr。我们可以使存储器成为同步写、同步读或者更复杂的端口结构，但是这种情况下，同步读和写操作将发生在时钟上升沿。而且，按照习惯，wr信号为低电平时表示写使能。由此得到SRAM的VHDL代码如下：

```
Architecture dualport of sram is
    Type sramdata is array (0 to 2**m-1) of
        Std_logic_vector (n-1 downto 0);
    Signal memory : sramdata;
Begin
    Process (clk ) is
    Begin
          If rising_edge(clk) then
```

146

```
        If wr = '0' then
            Memory(to_integer(unsigned(addr))) <= d;
        Else
            Q <= memory(to_integer(unsigned(addr)));
        End if;
      End if;
  End process;
End architecture dualport;
```

有几个有趣的地方值得注意。第一个是存储器的存取。如果我们将地址定义成了std_logic_vector类型，那么就不能简单地使用这个值来存取存储器数组中特定的元素。这一操作需要一个整数。也不能简单地将std_logic_vector类型直接转换为整数。必须先将std_logic_vector类型转换为无符号数。这是从std_logic_vector类型转换为整数的中间过程，因为我们可以将变量作为数字来使用，但是按照原始的std_logic_vector类型使用时则有位数的限制。很明显，这种情况下这并不是一个问题，因为我们并不想让地址超出存储器空间的大小，否则会出错。最后一步是将无符号数转换为整数。可以使用to_integer函数来完成这一步骤，这样就将地址转换为整数，从而访问存储器数组中特定的元素。

要使用这些数字类型转换函数，需要将IEEE标准库包含进模型中，如下所示：

```
Library ieee;
Use ieee.std_logic_1164.all;
Use ieee.numeric_std.all;
```

还有一点值得注意，读写功能是不能同时执行的，这样才能保证数据的完整性。模型的读写功能都由时钟控制，所以存储器的读写都是同步的。

## 10.6 Flash存储器

前面已经讨论过，从本质上说，Flash存储器是EEPROM（Electrically Erasable and Programmable Read Only Memory，电可擦除可编程只读存储器）的一种形式。与标准的RAM有些不同之处，标准的RAM中，给出地址后，依据R/W信号分别确定是读还是写。Flash存储器中典型的接口信号如下表所示。

147

| 引　脚 | 功　能 | 激 活 状 态 |
|---|---|---|
| CLE | 命令锁存 | 高电平有效，在WE上升沿激活 |
| ALE | 地址锁存 | 高电平有效，在WE上升沿激活 |
| CE | 片选信号 | 低电平有效 |
| RE | 读使能 | 下降沿 |
| WE | 写使能 | 上升沿 |
| WP | 写保护 | 低电平有效 |
| Busy | 准备好/忙 | 低电平为忙；高电平为准备好 |

除了这些控制信号外，当然还有地址总线和数据总线。为了实现这个模型，我们可以使用与标准RAM相似的实体：

```
ENTITY FLASH IS
  GENERIC (
    A : natural := 10;
```

```
    D : natural := 8
    );
  PORT (
    clk : in std_logic;
    addr : in std_logic_vector(A-1 downto 0);
    data : inout std_logic_vector (D-1 downto 0);
    cle : IN std_logic;
    ale : IN std_logic;
    ce : IN std_logic;
    re : IN std_logic;
    we : IN std_logic;
    wp : IN std_logic;
    busy : OUT std_logic;
    );
END ENTITY FLASH;
```

在大多数情况下，不需要对Flash存储器进行建模，只要有Flash的接口即可，所以描述Flash接口控制器的实体如下列代码所示：

```
ENTITY FLASHIF IS
  PORT (
    Clk : IN std_logic;
    read : IN std_logic;
    en : IN std_logic;
    cle : OUT std_logic;
    ale : OUT std_logic;
    ce : OUT std_logic;
    re : OUT std_logic;
    we : OUT std_logic;
    wp : OUT std_logic;
    busy : IN std_logic;
    );
END ENTITY FLASHIF;
```



这种器件的典型构造体定义如下：

```
Architecture basic of FLASHIF is
Begin
      Process (clk) is
        If busy = '1' then
          If rising_edge(clk) then
              Ce <= en;
              Ale <= '1';
              Cle < '1';
              If read = '0' then
                  We <= '1';
                  Re <= '1';
              Else
                  We <= '0';
                  Re <= '0';
```

```
                End if;
                If prog = '0' then
                     Wp <= '0';
                Else
                     Wp <= '1';
                End if;
            End if;
        End if;
    End process;
End architecture basic;
```

这只是Flash控制器的一个基本概况，不同的器件会有明显的差异。

## 10.7 小结

本章介绍了一些重要的存储器类型，如只读存储器、异步随机存取存储器、Flash存储器和同步随机存取存储器。但有一点必须记住，在大多数情况下，大容量的存储器通常都使用片外存储芯片，以上介绍的存储器模型完全是用来仿真的，而不能用于综合。不过在FPGA设计中，如果必须使用内部RAM，也可以少量使用。

这种情况下，速度与面积之间的权衡与取舍就显得尤其敏感，使用时必须非常仔细，千万不要天真地在FPGA中实现大量的存储器，因为这可能占用比实际需要多得多的存储器空间。

# 第11章　PS/2鼠标接口

## 11.1　引言

PS/2鼠标是一种标准接口，既可以用于计算机，又可以用于FPGA开发套件。PS/2协议是一个串行协议。本章将介绍该协议的基础知识，另外也编写了一个简单的主要用在标准FPGA开发套件上的VHDL接口代码，令设计者可以使用鼠标。

## 11.2　PS/2鼠标基础

PS/2鼠标的起源可以追溯到20世纪80年代IBM个人计算机（PC）的流行与普及。PS是Personal System（个人系统）的缩写，其第二版也因此称为PS/2，该接口及其名称一直沿用至今。

PS/2接口本质上是一种客户化的串行接口，每个连接器只支持一个设备（现代的USB系统中每个端口可以连接许多设备）。PS/2接口的数据率非常低，只有40kbit/s，设备使用5V直流电源供电。

USB设备一般都是可以"热插拔的"，随时都可以插入或拔出。而PS/2设备则不支持这样的特性，系统不关闭时不能移除。



PS/2鼠标支持鼠标与主机之间的相互通信，电源由主机通过一条5V电源线提供。

## 11.3　PS/2鼠标命令

PS/2鼠标的命令集非常有限，基本上就是按钮命令和移动命令。标准的鼠标有左、中、右按钮单击命令，以及*X*方向和*Y*方向的移动命令。*X*方向和*Y*方向的移动使用计数器来跟踪，计数值为相对于前一次鼠标发送数值的相对值，而不是绝对值。

## 11.4　PS/2鼠标数据包

PS/2鼠标以串行数据包的形式往数据线上发送数据，发送时用鼠标接口中的时钟线同步。每个数据包包含3个8比特的字节，第一个字节为配置字节，包括一些标志，第二个字节提供*X*方向的移动数据，第三个字节则提供*Y*方向的移动数据。具体描述如下表所示。

| 位 | 字节1 | 字节2 | 字节3 |
|---|---|---|---|
| 7 | *Y*方向溢出 | | |
| 6 | *X*方向溢出 | | |
| 5 | *Y*方向符号位 | | |
| 4 | *X*方向符号位 | *X*方向移动 | *Y*方向移动 |
| 3 | 总为1 | | |
| 2 | 中间按钮 | | |
| 1 | 右侧按钮 | | |
| 0 | 左侧按钮 | | |

每个移动数据字节由9比特二进制补码形式的数构成，符号位在字节1中定义。移动的范围为−255～+255。

## 11.5 PS/2操作模式

PS/2鼠标有4种基本的操作模式。通电后，鼠标进入“复位（reset）”模式，也可以由主机发送复位命令（0xFF）对鼠标进行初始化。复位完成之后，鼠标自动进入“流（stream）”模式。在这种模式下，数据由鼠标返回给主机。这两种模式是大多数应用中最常用的操作模式，不过还有另外两种模式［分别为“远程（remote）”模式和“回绕（wrap）”模式］。这两种模式主要为测试接口的操作是否正确而设置。 151

在复位模式中，鼠标进行复位并做一些基本的自检。然后定义一些默认的设置，例如采样周期为10ms，分辨率为每毫米4次计数，比例为1:1，数据报告功能不使能，等等。

鼠标发送设备ID（0x00）给主机，以便让主机知道这不是键盘或者某些复杂的鼠标，而是一个基本的PS/2鼠标。

一旦鼠标进入“流”模式，它将以定义好的采样速率发送数据包给主机，数据包内容为鼠标的活动，例如鼠标移动或者按钮动作。鼠标仅仅在有活动时才发送数据，否则它不做任何事。

如果主机请求鼠标进入“远程”模式，那么鼠标仅仅在主机请求时才发送数据，如果是在“回绕”模式，鼠标将每条命令原封不动地返回给主机（除了复位命令和复位回绕命令之外）。

## 11.6 PS/2滚轮鼠标

滚轮鼠标定义成了另一种类型的设备，其设备ID为0x03。使用这种鼠标时，复位之后，鼠标首先发送设备ID号，滚轮鼠标的数据包有4B，最后一个字节是滚轮运动数据。这一字节只使用低4bit（二进制补码形式），因此表示的范围是−8～+7。

## 11.7 基本PS/2鼠标处理模块VHDL代码

下面是用VHDL完成的最简单的PS/2鼠标处理程序，它使用鼠标的时钟信号作为系统时钟，对来自鼠标的数据进行监视，代码如下：

```
Library ieee;
Use ieee.std_logic_1164.all;

Entity psmouse is
       Port (
       Clock : IN std_logic;
       Data : IN std_logic
       );
End entity psmouse;
Architecture basic of psmouse is
       Signal d : std_logic_vector (23 downto 0);
       Signal byte1 ; std_logic_vector (7 downto 0);
```

152

```
        Signal byte2 : std_logic_vector (7 downto 0);
        Signal byte3 : std_logic_vector (7 downto 0);
        Signal index : integer := 23;
Begin
        Process(clock) is
        Begin
                If falling_edge(clock) then
                      D(index) <= data;
                      If index > 0 then
                              Index <= index-1;
                      Else
                              Byte1 <= d(23 downto 16);
                              Byte2 <= d(15 downto 8);
                              Byte3 <= d(7 downto 0);
                              Index <= 23;
                       End if;
                End if;
        End process;
End architecture basic;
```

这一VHDL代码非常简单。在时钟的下降沿，将data的当前值读入到数组d的下一个元素中。当读完24bit的数据之后（index减小到0），再将3B数据从数据包中转出来。

## 11.8 修改后的PS/2鼠标处理模块VHDL代码

前面的鼠标处理程序虽然在语义上没问题，但是在实际应用中却可能在鼠标时钟信号和数据信号上存在噪声，从而导致错误，因此另一种方法就应运而生。它使用更高的时钟频率来监视PS/2时钟，也就是将PS/2时钟看作一个普通信号。对PS/2时钟信号进行过滤，只有连续多次的值相同才认为是时钟发生变化。

```
Library ieee;
Use ieee.std_logic_1164.all;

Entity psmouse is
        Port (
             Clk : IN std_logic;
             Ps2_clock : IN std_logic;
             Data : IN std_logic
        );
End entity psmouse;
Architecture basic of psmouse is
       Signal clk_internal : std_logic := '0';
       Signal d : std_logic_vector (23 downto 0);
       Signal byte1 : std_logic_vector (7 downto 0);
       Signal byte2 : std_logic_vector (7 downto 0);
       Signal byte3 : std_logic_vector (7 downto 0);
       Signal index : integer := 23;
Begin
```

```
        Process(clock) is
              High : integer := 0;
              Low : integer := 0;
        Begin
                 If rising_edge(clock) then
                 if (ps2_clock = '1') then
                      if high = 8 then
                            clk_internal <= '1';
                            high <= 0;
                            low <= 0
                      else
                            high <= high + 1;
                      end if;
                 else
                      if low <= 8 then
                             clk_internal <= '0';
                             low <= 0;
                             high <= 0;
                      else
                             low <= low + 1;
                      end if;
                 end if;
                 End if;
  End process;
  Process(clk_internal) is
  Begin
          If falling_edge(clk_internal) then
                D(index) <= data;
                If index > 0 then
                      Index <= index - 1;
                Else
                      Byte1 <= d(23 downto 16);
                      Byte2 <= d(15 downto 8);
                      Byte3 <= d(7 downto 0);
                      Index <= 23;
                End if;
          End if;
  End process;
  End architecture basic;
```

这种情况下，修改后的程序在FPGA的更高频率的内部时钟驱动下，连续等待8个鼠标时钟信号的高电平或者低电平，然后才设置内部时钟为高或者低。处理数据输入的程序也具有同样的结构，只不过使用了内部产生的时钟。 [154]

## 11.9 小结

本章说明了如何处理基本的PS/2鼠标信号，然后存储3B的数据以备将来使用。还讨论了两种收集数据的方法：一种使用PS/2时钟，另一种使用更快的内部时钟进行采样。 [155]

# 第12章 PS/2键盘接口

## 12.1 引言

PS/2键盘是一种标准接口，既可以用于计算机，又可以用于FPGA开发套件。PS/2协议是一个串行协议，本章将介绍该协议的基础知识，另外也编写了一个简单的主要用在标准FPGA开发套件上的VHDL接口代码，令设计者可以使用PS/2键盘。

## 12.2 PS/2键盘基础

PS/2键盘的起源可以追溯到20世纪80年代IBM PC的流行与普及。键盘接口首先由XT（83个键，5引脚DIN）进化而来，经过AT（84到101个键，5引脚DIN），最终固定在PS/2（84到101个键，6引脚miniDIN）上。

PS/2接口本质上是一种客户化的串行接口，每个连接器只支持一个设备（现代的USB系统中每个端口可以连接许多设备）。PS/2接口的数据率非常低，只有40kbit/s，设备使用5V直流电源供电。

USB设备一般都是可以“热插拔的”，随时都可以插入或拔出。PS/2设备则不支持这样的特性，系统不关闭时不能移除。



PS/2键盘支持键盘与主机之间的相互通信，电源由主机通过一条5V电源线提供。

与鼠标不同，键盘上有一个处理器可以检查按键矩阵，以便确认按下的是哪个键，也可以发送适当数据到PS/2数据线上。

## 12.3 PS/2键盘命令

当某个键被按下时，键盘处理器发送给主机的命令有两条，分别是make和break。每个键在每种情况下都有独立的编码。实际发送给主机的编码与所发送字符的ASCII编码之间没有关系，主要是由主机对发送的键盘命令进行译码。例如，字符“5”的make编码为0x2E，break编码为0xF0和0x2E。大多数标准字符有1B的make编码和2B的break编码，扩展字符则经常有2B make编码和3B break编码。

如果有某个键被按下，键盘将周期性地发送make编码，直到另一个键被按下。发送的速率称为重复率（typematic rate），默认值为每秒大约10个字符。

## 12.4 PS/2键盘数据包

PS/2键盘以串行数据包的形式往数据线上发送数据，发送时用键盘接口中的时钟线同步。每个数据包包含3个8比特的字节，可以通过键盘扫描编码查找表进行译码。

# 12.5 PS/2键盘操作模式

## 12.5.1 基本PS/2键盘处理模块VHDL代码

下面是用VHDL完成的最简单的PS/2键盘处理程序，它使用键盘的时钟信号作为系统时钟，对来自键盘的数据进行监视，代码如下：

```
Library ieee;
Use ieee.std_logic_1164.all;

Entity pskeyboard is
       Port (
              Clock : IN std_logic;
              Data : IN std_logic
       );
End entity pskeyboard;
Architecture basic of pskeyboard is
      Signal d : std_logic_vector (23 downto 0);
      Signal byte1 : std_logic_vector (7 downto 0);
      Signal byte2 : std_logic_vector (7 downto 0);
      Signal byte3 : std_logic_vector (7 downto 0);
      Signal index : integer := 23;
Begin
      Process(clock) is
      Begin
            If falling_edge(clock) then
                  D(index) <= data;
                  If index > 0 then
                        Index <= index-1;
                  Else
                        Byte1 <= d(23 downto 16);
                        Byte2 <= d(15 downto 8);
                        Byte3 <= d(7 downto 0);
                        Index <= 23;
                  End if;
            End if;
      End process;
End architecture basic;
```

157

这一VHDL代码非常简单。在时钟的下降沿，将data的当前值读入到数组d的下一个元素中，当读完24bit的数据之后（index减小到0），再将3B数据从数据包中转出来。

## 12.5.2 修改后的PS/2键盘处理模块VHDL代码

前面的键盘处理程序虽然在语义上没问题，但是在实际应用中却可能在键盘时钟信号和数据信号上存在噪声，从而导致错误。因此另一种方法就应运而生，它使用更高的时钟频率

来监视PS/2时钟，就是将PS/2时钟看作一个普通信号。对PS/2时钟信号进行过滤，只有连续多次的值相同才认为是时钟发生变化。

158

```
Library ieee;
Use ieee.std_logic_1164.all;

Entity pskeyboard is
       Port (
             Clk : IN std_logic;
             Ps2_clock : IN std_logic;
             Data : IN std_logic
       );
End entity pskeyboard;
Architecture basic of pskeyboard is
      Signal clk_internal : std_logic := '0';
      Signal d : std_logic_vector (23 downto 0);
      Signal byte1 : std_logic_vector (7 downto 0);
      Signal byte2 : std_logic_vector (7 downto 0);
      Signal byte3 : std_logic_vector (7 downto 0);
      Signal index : integer := 23;
Begin
      Process(clock) is
            High : integer := 0;
             Low : integer := 0;
      Begin
            If rising_edge(clock) then
            if (ps2_clock = '1') then
                  if high = 8 then
                         clk_internal <= '1';
                         high <= 0;
                         low <= 0;
                  else
                         high <= high +1;
                  end if;
            else
                  if low = 8 then
                         clk_internal <= '0';
                         low <= 0;
                         high <= 0;
                  else
                         low <= low +1;
                  end if;
            end if;
      End if;
      End process;
Process(clk_internal) is
```

```
Begin
      If falling_edge(clk_internal) then
            D(index) <= data;
            If index > 0 then
                  Index <= index-1;
            Else
                  Byte1 <= d(23 downto 16);
                  Byte2 <= d(15 downto 8);
                  Byte3 <= d(7 downto 0);
                  Index <= 23;
            End if;
      End if;
End process;
End architecture basic;
```

这种情况下，修改后的键盘处理程序在FPGA的更高频率的内部时钟驱动下，连续等待8个键盘时钟信号的高电平或者低电平，然后才设置内部时钟为高或者低。处理数据输入的程序也具有同样的结构，只不过使用了内部产生的时钟。 159

## 12.6 小结

本章说明了如何处理基本的PS/2键盘信号，然后是如何存储3B的数据以备将来使用。讨论了两种收集数据的方法，一种使用PS/2时钟，另一种使用更快的内部时钟进行采样。 160

# 第13章　一个简单的VGA接口

## 13.1　引言

绝大多数现代计算机的显示器普遍采用VGA（Video Graphics Array）接口，其技术基础是像素映射、颜色平面以及水平和垂直同步信号。VGA显示器有3个颜色信号，分别为红、绿、蓝，设置这些颜色的开与关可以控制屏幕的显示。这些颜色不同亮度的组合决定了屏幕上最终所显示的颜色。例如，如果红色全部打开，而蓝色和绿色关闭，那么最终看到的颜色是红色。每种颜色的模拟亮度用两比特数字量表示（例如red0和red1），将这些数字量连接到数模转换器就可以得到正确的输出信号。

屏幕的分辨率可以从480×320像素到更大，不过标准的默认屏幕分辨率为640×480像素，也就是整个屏幕共有480行，每行含有640个像素，因此屏幕高宽比是640/480，这是传统显示器屏幕的经典设置。

VGA图像由两个信号控制：水平同步信号和垂直同步信号。水平同步信号使用一个负脉冲表示一行像素的开始和结束。实际的图像数据是在25.17μs的时间窗口内发送的，而同步脉冲之间的间隔为31.77μs。没有图像数据发送的这段时间定义为消隐区，此时的图像为黑色。垂直同步信号与水平同步信号相似，只不过其负脉冲表示整个一帧图像的开始与结束，一帧图像的时间为15.25ms，而垂直脉冲之间的间隔为16.784 ms。



脉冲之间的数据间隔有一些约束要求，本章后面将讨论这些问题。不过显而易见的是，要想得到一个正确的VGA输出，其关键问题是要有精确的时序与数据定义（这里使用VHDL）。

## 13.2　基本像素时序

如果有25.17μs的时间间隔来处理所需的像素，那么必须做一些计算，以确保FPGA可以在一定的时间内正确地显示图像。例如，如果有一个640×480像素的VGA系统，那么640像素必须在25.17μs内发送给显示器。经过简单的计算可知，每个像素需要的时间是25.17μs/640 = 39.328 ns。如果FPGA的时钟频率是100 MHz，也就是说最小的时钟周期为10 ns，那么用标准的FPGA即可满足这一时序要求。

## 13.3　图像处理

很明显，在FPGA中实现整个图像处理系统是不明智的，但是将图像存储在RAM中、然后一帧一帧地进行恢复比较可取。因此，VGA接口对于在存储器内移动图像数据具有重要意义，同时，使用前面定义的RAM接口也很有意义。所以，除了VGA接口外，VGA控制器还应该包含一个RAM接口。

## 13.4 VGA接口的VHDL实现

定义VGA接口的第一步是创建一个VHDL实体，这个实体具有全局的时钟和复位信号以及VGA输出管脚和存储器接口。粗略的VHDL实体如下列代码所示：

162

```
Library ieee;
Use ieee.std_logic_1164.all;
Entity vga is
  Port (
        Clk : IN std_logic;
        Nrst : IN std_logic;
        Hsync : OUT std_logic;
        Vsync : OUT std_logic;
        Red : OUT std_logic_vector (1 downto 0);
        Green : OUT std_logic_vector (1 downto 0);
        Blue : OUT std_logic_vector (1 downto 0);
        Address : OUT (std_logic_vector (15 downto 0);
        Data : IN (std_logic_vector (7 downto 0);
        RAM_en : OUT std_logic;
        RAM_oe : OUT std_logic;
        RAM_wr : OUT std_logic
  );
End entity vga;

Architecture core of vga is
  -- VGA internal signals go here
Begin
  -- VGA Interface core goes here
End architecture core;
```

构造体包含许多过程（process），还有一些内部信号用来管理像素数据从存储器到屏幕的传输。从实体中可以看出，从存储器中读出的数据是8bit的数据块，而我们需要3×2bit表示一个像素，因此当返回数据的时候，每个字节包含一个像素的数据。在这个例子中，由于我们使用640×480像素的图像，所以最少需要307 200B的存储器空间。这就是说，如果不对图像进行处理而直接使用存储器存放，每兆字节才能存放3帧图像。当然，在实际应用中，通常会对图像进行某种压缩处理（例如JPEG），但是这已经超出了本书的范围。

因此，我们可以使用一个简单的过程（process）从存储器内得到当前像素的数据，代码如下：

```
Mem_read : process ( pclk, nrst ) is
      signal current_address : unsigned (16 downto 0);
Begin
      If nrst = '0' then
            Pixelcount <= 0;
            Current_address <= 0;
      Else
            If rising_edge(pclk) then
```

```
                Current_address <= current_address + 1;
                Address <= std_logic_vector
                  (current_address);
                Pixel_data <= data;
             End if;
          End if;
       End process;
```

163

这一过程将当前像素的值返回给信号pixel_data，这个信号是在构造体层声明的：

```
signal pixel_data : std_logic_vector ( 7 downto 0 );
```

在这个信号中，8bit数据的低6bit分别表示红（red）、绿（green）和蓝（blue），其位号分别为0–1、2–3和4–5。

## 13.5 水平同步

下一个关键的过程是水平和垂直同步脉冲信号的时序以及其间的消隐间隔。VGA的行时序为：每行时间是31 770ns，其中显示数据的时间窗口为25 170ns。如果FPGA运行在100MHz（周期为10ns），那么意味着每行需要3177个时钟周期，其中2517个时钟周期用于显示像素数据，总共660个消隐脉冲（左右各有330个）。这也意味着，对于640个像素的行来说，每个像素需要39.3ns。我们可以将这个时间取整，得到每个像素4个时钟周期。读者可能已经注意到了，对于重建像素，我们使用了一个新的内部时钟信号pclk，可以利用这个时序信息在内部产生适当的像素时钟（pclk）。

这稍微加大了显示窗口的宽度，因此消隐脉冲必须减小到617个时钟周期，也就是说显示窗口之前为308个时钟周期，之后为309个时钟周期。

从另一个方面说，水平同步脉冲发生在整个间隔的26 110 ns到29 880 ns之间。这比整个行的时间少了189个时钟周期，所以水平同步脉冲在94个时钟周期之后变为低电平，然后在行结束之前95个时钟周期再变为高电平。水平同步脉冲的外部和内部时序之间的差异是377个时钟周期，因此同步脉冲必须在显示窗口之前的(94+188)个时钟周期变为高电平，然后在显示窗口前(95+189)个时钟周期变为低电平。

水平同步脉冲信号具有如下的行为特性。

| 时钟周期 | 值 |
|---|---|
| 0 | 1 |
| 94 | 0 |
| 282 | 1 |
| 2893 | 0 |
| 3082 | 1 |

164

这可以通过一个简单的计数器来实现：

```
Hsync_counter : process ( clk, nrst ) is
      Hcount : unsigned ( 11 downto 0 );
Begin
      If nrst = '0' then
            Hcount <= 0;
            Hsync <= '1';
```

```
        Else
            If hcount > and hcount < 2988 then
                hsync <= '0';
            else
                hsync <= '1';
            End if;
            If hcount < 3177 then
                Hcount <= hcount + 1;
            Else
                Hcount <= 0;
            End if;
        End if;
    End process;
```

## 13.6 垂直同步

水平同步管理着每一行中的各个像素，垂直同步则将每一行看作一个整体生成一幅图像。一帧（包含所有行）的周期为16 784 000ns。在这段时间之内，图像的所有行要在15 250 000 ns内显示完毕。剩下的时间为垂直消隐间隔，垂直同步脉冲在15 700 000ns时变为低电平，到15 764 000 ns时再变为高电平。

显然，为这些计算定义周期为10ns的时钟并非明智之举，由于时钟频率最大的公约数是2μs，所以我们可以将系统时钟频率除以2000，得到周期为2μs的垂直同步时钟，以便使设计尽可能简单并且紧凑。

```
    Clk_div : process ( clk, nrst ) is
    Begin
        If nrst = '0' then
            Count <= 0;
            Vclk <= '0';
        Else
            If count = 1999 then
                Count <= 0;
                Vclk <= not vclk;
            Else
                Count <= count + 1;
            End if;
        End if;
    End process;
```

在这里，垂直同步时钟vclk定义为std_logic类型。在下一个过程中可以用这个信号控制vsync脉冲的产生，垂直同步脉冲源自时钟信号：

```
    Vsync_timing : process (vclk) is
    Begin
        If nrst = '0' then
            Vcount <= 0;
        Else
```

```
            If vcount>15700 and vcount < 15764 then
                  Vsync <= '0';
            Else
                  Vsync <= '1';
            End if;
            If vcount > 16784 then
                  Vcount <= 0;
            Else
                  Vcount <= vcount + 1;
            End if;
      End if;
End process;
```

利用这个过程，就产生了垂直同步（帧同步）脉冲。

## 13.7 水平和垂直消隐脉冲

除了基本的水平和垂直同步脉冲计数器外，还必须定义一个水平消隐脉冲，用来设置在25 170ns（2517个时钟周期）后将行数据变为低电平。这可以用一个简单的计数器来实现，方法与水平同步脉冲完全一样，垂直消隐脉冲也类似。以下用VHDL描述的两个过程实现了这些功能。

```
Hblank_counter : process ( clk, nrst ) is
      Hcount : unsigned ( 11 downto 0 );
Begin
      If nrst = '0' then
            Hcount <= 0;
            hblank <= '1';
      Else
            if hcount > 2517 and hcount < 3177 then
                  hblank <= '0';
            else
                  hblank <= '1';
            End if;
            If hcount < 3177 then
                  Hcount <= hcount + 1;
            Else
                  Hcount <= 0;
            End if;
      End if;
End process;
Vblank_timing : process (vclk) is
Begin
      If nrst = '0' then
            Vcount <= 0;
            Vblank <= '1';
      Else
            If vcount > 15250 and vcount < 16784 then
```

```
            vblank <= '0';
        Else
            vblank <= '1';
        End if;
        If vcount > 16784 then
            Vblank <= 0;
        Else
            Vcount <= vcount + 1;
        End if;
    End if;
End process;
```

## 13.8 计算正确的像素数据

前面我们已经看到，像素数据是在像素时钟（pclk）的驱动下由存储器中恢复出来的。像素时钟是由系统时钟4分频后得到的。在pclk的每个上升沿，下一个像素数据由存储器中得到，相应的信号为data，并要转换为红、绿和蓝的行信号。这可以通过下面的过程处理：

```
Pixel_handler : process (pclk) is
Begin
    Red <= data(1 downto 0);
    Green <= data(3 downto 2);
    Blue <= data(5 downto 4);
End process;
```

这是一个基本的处理过程（process），可以选出正确的像素数据，但是不包含视频消隐信号。如果要包含视频消隐信号，相应的VHDL代码如下： 167

```
Pixel_handler : process (pclk) is
    Blank : std_logic_vector (1 downto 0);
Begin
    Blank(0) <= hblank or vblank;
    Blank(1) <= hblank or vblank;
    Red <= data(1 downto 0) & blank;
    Green <= data(3 downto 2) & blank;
    Blue <= data(5 downto 4) & blank;
End process;
```

这是最后一步，到这里我们就完成了基本的VHDL VGA处理模块。

## 13.9 小结

本章介绍了用VHDL语言开发简单的VGA处理模块的基础知识。这只是单纯从处理的角度看问题，不过希望读者明白如何使用不太复杂的VHDL语句和构造块来开发VGA接口。为特定显示器开发完整的VGA接口留给读者去实现，可以利用本章中的技术作为基础。 168



# 第四部分　优化设计

这一部分将介绍一些“高级”的主题。本书其他部分强调的是“做什么”，而这一部分则更加关注“如何做”。如何对设计进行综合？如何使我们的设计面积更小而速度更快？如何与实际应用中的混合信号系统进行接口？如何开发可验证设计？所有这些设计挑战都将在本部分中得到解答。

# 第14章 综　　合

## 14.1 引言

VHDL最初的设计目的是为数字电路提供一种设计规范语言。这一工作的主要目标，是使设计描述能够被仿真以验证设计是否满足规范要求。当VHDL被标准化为IEEE Std 1076后，不仅仅在仿真方面，在硬件设计流程的主要部分中更加广泛的应用也成为可能。

最初，设计数字电路的方法主要是基于电路图的，然后使用门级元件库直接从电路图产生寄存器传输级（Register Transfer Logic，RTL）网表。很明显，当设计规模相对较小时，这是一种合理的设计技术。但是很快，情况就出现了变化。对于现代的拥有数百万门的FPGA来说，任何大小的设计规模都不适合用这种电路图的方式。

众多EDA厂商在VHDL发展的早期就清楚地认识到，如果有一种标准的语言能够描述数据流程和控制流程，那么就有可能从某种更高的层次描述自动生成门级的VHDL设计，寄存器传输级（RTL）显然是一个开始点。RTL可以直接对数据流程和控制流程进行描述，也能够很容易地映射到标准门级逻辑上。随后出现的综合软件（例如，Synopsys公司的Design Compiler）很快就在ASIC和FPGA数字设计流程中扮演了一个很重要的角色，并事实上成为数字设计工程师效率提升的一个强大的驱动力。如果没有RTL综合，现代的高密度设计是不可能的。

171

正是基于此，现代的设计人员经常将“RTL综合”简称为“综合”，但这远不是全部。由于设计变得越来越复杂，对于行为综合的需求也在不断增加，然而各大EDA厂商对行为综合软件的支持却大相径庭。

## 14.2 RTL综合支持的VHDL

当VHDL被标准化后，综合还没有标准化，因此可综合VHDL只是整个VHDL语言的一个子集。设计者面临的另一个普遍问题是，不同的综合软件对同样的VHDL输入进行综合，却给出不同的综合结果，甚至在某种条件下，有些综合软件能够综合，有些却完全不能综合。

有一些标准的VHDL技术不能综合，本章将会对此进行总结。

VHDL中有两种因素不支持综合，一种引起综合失败，另一种则被综合软件忽略。引起综合失败的因素在许多方面都容易管理，因为综合软件会提示一些错误信息。被忽略的因素则具有更大的隐蔽性，因为它们会在综合后的设计中留下一些错误，可能直到测试硬件时才能发现这些错误。

### 14.2.1 初始条件

VHDL支持对信号和变量设置初始条件，即初值，但这是不能进行物理实现的。在实际设计中，综合后的电路的初始条件为随机值，因此应该总是一个外部复位管脚对内部逻辑复

位，这是因为在综合过程中，初始条件会被综合软件忽略。

### 14.2.2 并发边沿

通常情况下，人们会使用时钟边沿作为一个模型的触发开关，因此一个简单的VHDL模型中可能会有一个如下所示的过程，它会等待时钟的上升沿：

```
Process (clk)
    If rising_edge(clk) then
        Q <= q;
    End if;
End process;
```

172

或者用另一种类似的方法：

```
Process (clk)
    If clk'event and clk = '1' then
        Q <= q;
    End if;
End process;
```

如果在触发条件中有多于一个的时钟上升沿，那么该触发就是无效的：

```
Process (clk1, clk2)
    If rising_edge(clk1) and rising_edge(clk2) then
        Q <= d;
    End if;
End process;
```

这将会令综合失败。

### 14.2.3 数字类型

综合仅仅支持范围有限的数字。例如，未定义范围（即无限的）的整数类型就不被综合软件支持。就一般的情况来讲，在综合以前通常需要设计者指定整数的范围以及其他基于整数的数字（例如有符号数和无符号数）。

对于向量而言（用数字作为索引），有一个微妙的限制，就是该索引值必须事先定义，因此总线宽度不能是变量。

综合软件一般不支持浮点数（实数），因为软件本身没有定义浮点类型库。

### 14.2.4 wait语句

只有wait语句以隐含敏感列表的形式出现，并且具有确定的值，它才会被综合软件支持。所以，如果wait语句具有如下形式：

```
Wait on clk = '1';
```

那么综合软件将支持这种语句。如果wait语句描述了一段时间的延迟，那么综合软件不会支持。例如，如下的语句综合软件无法支持：

173

```
Wait for 10 ns;
```

### 14.2.5 断言

任何形式的断言（assertion）都会被综合软件忽略。

### 14.2.6 循环

for循环是VHDL中循环的一种特殊情况，综合软件要求循环的次数必须定义为全局静态值。这就是说，在综合时不能使用变量来定义可能会改变的for循环次数。

如果要实现一个while循环，那么在循环体内必须有一个wait语句，否则可能会进入死循环。

## 14.3 一些引起综合失败的情况

各种综合软件之间有一些差异，因此在使用时必须倍加小心，以便保证各个综合软件之间的协同工作能力，尤其是在多团队设计或者使用第三方VHDL内核时。从第三方得到的内核可能使用了与当前设计流程不同的综合软件进行综合，所以宣称的“可综合”内核可能在你的设计流程中并不总是可综合的。

正因为如此，通常要尽可能保持VHDL代码的通用性，而且如果打算发布一些IP核或者使用不同的综合工具，那么就要避免使用某些特殊包中的技巧。这可能会使VHDL代码不够简洁，但是其可靠性却更强，而且可以避免一些潜在的问题（这些问题可能会引起后续设计流程出现重大延迟，尤其是在集成阶段）。

一种情况是在同一个过程中使用不同的触发变量。例如，如果有一个时钟信号和一个复位信号，或者一个时钟信号和一个使能信号，那么很容易就会将逻辑合并到一个表达式中，如下所示：

```
If (clk'event and clk = '1' and nrst = '1') then
   . . .
End if;
```

然而，对于某些综合软件而言，这将引起错误。将这些变量分散到嵌套的if语句中更可取，原因如下：（1）代码可读性更好；（2）未定义逻辑状态出现的可能性更小；（3）综合软件不会因为这样的VHDL代码出现问题。

174

## 14.4 综合的内容

### 14.4.1 总体设计结构

综合数字电路的基本方法是将每一个设计模块看作控制器和数据路径的组合。其中的控制器通常是一个有限状态机（FSM），受时钟控制。数据路径多为组合逻辑电路，但是也可能有一些存储结构，因此也需要时钟信号。基本框图如图14-1所示。

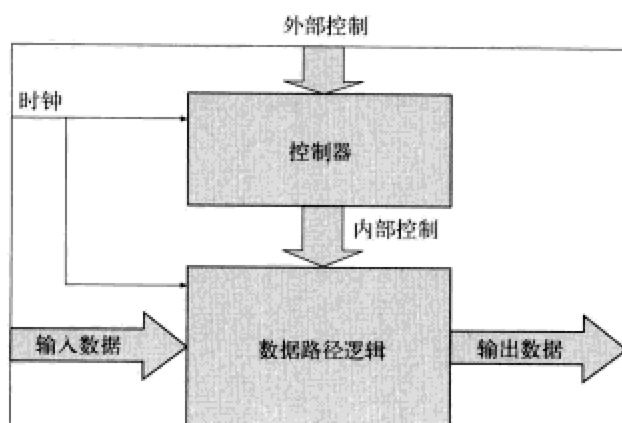


图14-1 可综合数字电路

## 14.4.2 控制器

控制器用来为数据路径产生控制信号，也可能有一些外部的控制信号，因此一般情况下既有内部控制信号，又有外部控制信号。由于这是一个有限状态机，所以整个设计是同步的，需要时钟驱动，一般还有一个复位信号。

控制器可以用状态图（或气泡图）来表示。其中显示了每一个独立的状态以及各个状态之间的转换。控制器有两种类型：Moore型（状态机输出完全依赖于状态变量）和Mealy型（状态机输出不但与当前状态有关，而且还与输入有关）。状态机的行为可以用状态图（有时也称为状态表）来表示，如图14-2所示。

175

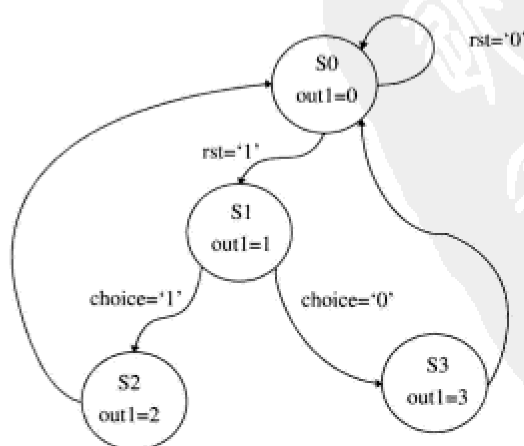


图14-2 基本状态机

有限状态机的相关技术将在本书后面的章节中描述，但这里需要记住的关键一点是，之所以称为有限状态机，是因为只有有限个状态，因此存储单元（D型触发器）的数量在状态机的定义中也就隐含地确定了。另外，VHDL语言允许将状态名称定义为枚举类型，这样VHDL代码可读性好，容易理解，也易于综合。

例如，有一个简单的状态机，只有两个状态，分别定义为ON和OFF。如果激励信号为低，状态机进入OFF态；如果激励信号为高，状态机进入ON状态。

为了用VHDL实现这个简单的状态机，可以使用一种新的类型来表示状态：

```
Type states is (OFF, ON) ;
Signal current_state, next_state : states;
```

注意，在状态机的VHDL代码中，我们已经定义了当前状态和下一个状态。在一个过程中使用case语句就可以很容易地实现状态机的主体部分，这个过程会等待信号current_state以及外部变量或者控制信号的变化：

```
Process (current_state, onoff)
Begin
     Case current_state is
          When OFF =>
               If onoff = '1' then
                    Next_state <= ON;
               End if;
          When ON =>
               If onoff = '0' then
                    Next_state <= OFF;
               End if;
     End case;
End process;
```

176

在构造体的其他地方，需要用下一个状态的值赋值给current_state，代码如下：

```
Current_state <= next_state;
```

### 14.4.3 数据路径

所谓数据路径，从名称猜测就是对输入数据进行处理并产生正确的输出数据的逻辑。数据路径按照功能通常可以分为几个模块，这就为速度和面积的优化提供了可能。例如，如果面积不是问题，而速度是主要的关注对象，那么就可以构造一个较大的设计，尽可能在一个时钟周期内产生输出。如果面积不是问题，而吞吐率必须大，那么可以使用流水线技术使总的数据率达到最大，但是单次操作的延迟却增加了。最后，如果面积是一个关键因素，那么可以用一个功能模块，然后用寄存器存储中间结果，重复执行同一个功能即可。很明显，这种实现方式的速度更慢，但是面积却更小。

177

基本的数据路径模型中，组合逻辑模块被几个寄存器所隔开（见图14-3）。显然，优化数据流程时有一些选项，可以决定在速度和面积优化时如何在寄存器之间移动数据。

重要的一点是，一定要遵守某些简单的规则才能保证综合过程顺利进行。首要规则就是，要确保组合逻辑中的每一个信号在每次循环中都有定义，换句话说，就是一定不要在case或者if语句中留下未定义的分支。如果出现这种情况，那么综合软件会推断出一个锁存器，而锁存器被综合后不会与全局时钟相连，因此其行为是不可预测的。

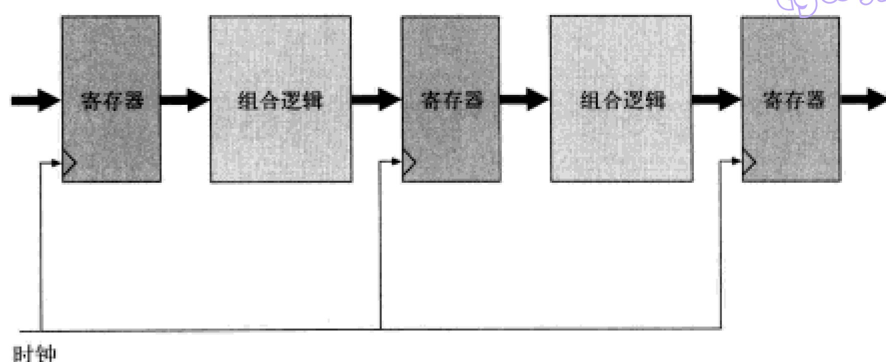



图14-3 数据路径

## 14.5 小结

本章介绍了综合的概念，既从设计者的角度分析，又考虑了出于综合的目的而使用某种类型的VHDL的问题。本章还描述了各种方法的假设前提和局限性，定义了一些明智的实践方法，以便获得更加实用的设计。 178

# 第15章 VHDL行为建模

## 15.1 引言

在许多设计中，对更高抽象级的要求日益迫切，以便使整个系统级设计更加容易。如果设计可以由行为描述，尤其是当综合方法可以自动处理任何有关时钟、区块划分或者实现的问题时，在系统级就很少有必要考虑实现的细节问题。

而且，通过系统级或者行为级的分析，可以在设计过程的早期就制定决策，这样可以避免一些可能造成重大损失的错误。使用这种方法还可以初步估计出面积和功耗的大小，也可以制定关键的性能规范和架构决策，而这一切根本不需要有每个模块的详细设计。

## 15.2 怎样从RTL转向行为级

从某种意义上来说，从RTL到行为级的转移是非常直接的，因为VHDL本身其实很简单。没有必要保证时钟的正确性，也没有必要保证各个独立的过程（process）为构造体的不同区域作实现，甚至没有必要实例化各个元件。

举一个实际的例子可能有助于说明这一点。例如有一个向量积乘法器，我们可以看一看其RTL描述和行为级描述之间有什么差异。首先给出RTL的描述，然后再说明如何抽象出一个行为级模型。第一步先说明模型的规格，如图15-1所示。



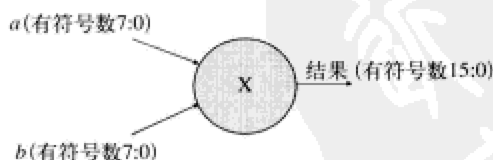



图15-1 向量积乘法器规格

模型的数据路径如图15-2所示。

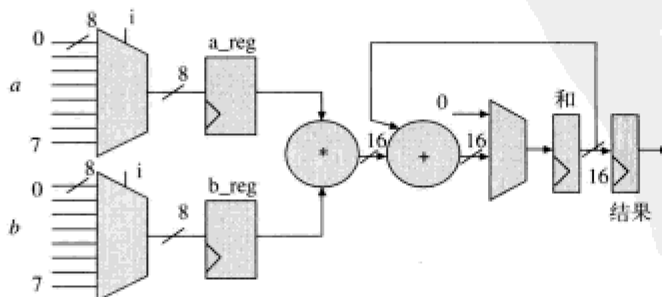



图15-2 模型的数据路径

第一个任务是定义模型的VHDL实体类型，如下面的代码所示。注意，我们已经定义过一种新的类型sig8，这是一种有符号向量类型，由此，定义向量积乘法器如下：

```
library ieee;
Use ieee.std_logic_1164.all;
Use ieee.numeric_std.all;
Package cross_product_types is
     subtype sig8 is signed (7 downto 0);
     type sig8_vector is array
           (natural range<>) of sig8;
End package;
```

现在也将实体放在一起，如下面的代码所示。RTL结构中，我们需要时钟和复位信号。

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.cross_product_types.all;

entity cross_product is
    port(
        a,b : in sig8_vector(0 to 7);
        clk, reset : in bit;
        result : out signed(15 downto 0)
    );
end entity cross_product;
```

180

下面设置基本的构造体，它具有基本的内部信号定义。过程将分别进行说明。

```
architecture rtl of cross_product is
  signal I : unsigned (2 downto 0);
  signal ai, bi : sig8;
  signal product, add_in, sum, accumulator : signed (15
    downto 0);
begin
       control: process (clk)
       begin
         if clk'event and clk = '1' then
           if reset = '1' then
             i <= (others => '0');
           else
             i <= i + 1;
           end if;
         end if;
       end process;
       a_mux : ai <= a(i);
       b_mux <= bi <= b(i);
       multiply : product <= ai * bi;
       z_mux : add_in <= X"000" when i = 0 else
```

```
    accumulator;

  accumulate : process (clk)
  begin
    if clk'event and clk = '1' then
      accumulator <= sum;
    end if;
  end process;

  output : result <= accumulator;
end;
```

这里需要注意，有两个过程（process），一个进行累加，另一个处理乘法运算。重要的一点是，后续的步骤不是显而易见的。即使在这个简单的模型中，也很难将状态机的主要行为提取出来。在复杂的控制器中，这一任务几乎是不可能的，除非我们对设计结构非常熟悉。当使用任何综合工具处理任意层次的VHDL或Verilog代码时，这一点尤为重要。

现在，重新考虑使用行为级VHDL。行为级模型使用与RTL模型相同的包（package）和库（library），不过要注意没有明显地定义时钟和复位。

181

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.cross_product_types.all;
entity cross_product is
   port(
       a,b : in sig8_vector(0 to 7);
       result : out signed(15 downto 0)
   );
end entity cross_product;
```

在这个模型中，构造体更加简单，可以用比RTL直接得多的方法进行建模。

```
architecture behav of cross_product is
begin

  process
    variable sum : signed(15 downto 0);
  begin
    sum := to_signed(0,16);
    for i in 0 to 7 loop
      sum := sum + a(i) * b(i);
    end loop;
    result <= sum;
    wait for 100 ns;
  end process;

end architecture;
```

观察模型的功能更加容易，而且调试也比在RTL模型中要简单得多。总体来说，设计清

晰明了，代码可读性好，功能也容易确定。需要注意的是，这里并没有明显的控制器，不过综合工具将会定义适当的机制。同时也要注意，模型仅仅用了一个过程（process）定义。综合工具将根据给定的优化约束对设计进行划分。

还要注意一下等待语句（wait）。这一语句为系统引入了一个隐含的时钟延迟。很明显，这将依赖于实际中使用的时钟。另外，还有一个隐含的复位信号。如果需要定义一个显式的时钟，那么可以使用wait until rising_edge（clk）或者类似的方法，不过最好保持模型的行为级本色。

考虑另一个有用的例子：有限脉冲响应（Finite Impulse Response，FIR）滤波器。先不考虑实体（entity）和声明（declaration），那么我们怎么用VHDL构造一个理想的FIR滤波器行为级模型呢？

接口规范定义如下：

```
Input : signed (15 downto 0)
Output : signed (15 downto 0)
Coefficients : array(natural range<>) of integer...
```

182

最终的VHDL代码如下所示：

```
for I in samples'right downto 1 loop
      samples(I) := samples(I-1);
end loop
samples(0) := input;

sum := to_signed(0,32);
for j in 0 to samples'right loop
      sum := sum + (to_signed(coeffs(j),16) *
        samples(j));
end loop;

output <= sum(30 downto 15);
wait for 1 us;
```

这一模型很容易进行参数化配置，也容易修改，而且结构清晰，易于理解。

## 15.3 小结

对于最初的设计想法，行为级VHDL是一项有用的技术，同时也是RTL设计的一个良好起点。但是，一定要记住，相当多的行为级VHDL不能综合，完全用于概念设计或者用在测试平台中。为了使行为级VHDL成为有用的设计工具，设计者可以利用VHDL具有建立多个构造体的能力，使用同样的测试平台验证RTL模型，同时也验证行为模型，保证设计的正确性。

总而言之，我们可以在早期使用行为级模型，对以下几个方面有重要的意义：

- ❑ 执行快速的功能仿真；
- ❑ 制定性能标准/设计的权衡取舍；
- ❑ 研究非线性效应；
- ❑ 检查实现的具体事项；
- ❑ 执行拓扑评估。

183

# 第16章　设计优化

## 16.1　引言

所谓设计优化，就是为了使某个设计在优化后比原始设计的性能有大幅度的提高。在数字设计，尤其是FPGA设计中，设计优化通常具有三大指标，即速度、面积和功耗。在讨论这些优化目标的实施细节以前，我们先讨论一些原理性的内容，以便让读者明白在综合时都发生了些什么。

当使用VHDL进行FPGA设计时，设计优化通常包含两个方面的内容，一个是寄存器传输级（RTL）的优化，最终得到优化后的VHDL逻辑表达式。另一个是在映射底层函数到独立的工艺门级电路前对基本逻辑进行最小化处理。

## 16.2　逻辑优化技术

有两种方式可以使设计中的逻辑达到最小，一种是保持层次关系，另一种是将电路打平，不再具有模块间的层次关系。综合工具通常允许用户选择必要的综合选项。显然，打平设计的优点是，设计可以被看成一个整体，而保持层次关系，则设计将具有结构化的特点，有利于分析电路整体的行为。



逻辑最小化的基本方法是将逻辑等式简化为两级形式（即乘积和的形式）。对于简单的设计，最常用的方法是使用卡诺夫图，将输入和输出变量标注在卡诺夫图中，就可以得到输出表达式，并且该表达式要比原始的布尔型表达式更小。

例如，一个基本的4输入卡诺夫图，如图16-1所示。

| CD \ AB | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | $Z_0$ | $Z_4$ | $Z_8$ | $Z_{12}$ |
| 01 | $Z_1$ | $Z_5$ | $Z_9$ | $Z_{13}$ |
| 11 | $Z_2$ | $Z_6$ | $Z_{10}$ | $Z_{14}$ |
| 10 | $Z_3$ | $Z_7$ | $Z_{11}$ | $Z_{15}$ |

图16-1　基本4输入卡诺夫图

当逻辑表达式使用逻辑等式表示时，我们可以在所有为1的方格中画圆圈，选择所有有效输出，这样就定义了基本的逻辑行为。具体方法是使圆圈尽可能大，包含尽可能多的1，并且包含尽可能少的输入变量。例如，如果一个逻辑等式定义为 $Z=A\cdot B\cdot\overline{C}+\overline{A}\cdot B\cdot D+\overline{A}\cdot B\cdot D$，那么相应的卡诺夫图如图16-2所示。

| CD \ AB | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 0 | 0 | 1 | 0 |
| 01 | 0 | 1 | 1 | 0 |
| 11 | 0 | 1 | 0 | 0 |
| 10 | 0 | 0 | 0 | 0 |

图16-2 具体的卡诺夫图例子

现在，如果直接进行物理实现，需要3个3输入与门、一个3输入或门以及几个非门。从卡诺夫图中可以看出，尽管我们只定义了两个逻辑功能，但是在原始定义中有冗余表达式，我们可以将其简化为图16-3所示的两个输入的组合逻辑，两者的输出是相同的。

185

| CD \ AB | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 0 | 0 | 1 | 0 |
| 01 | 0 | 1 | 1 | 0 |
| 11 | 0 | 1 | 0 | 0 |
| 10 | 0 | 0 | 0 | 0 |

逻辑表达式

隐含逻辑

图16-3 用卡诺夫图确定功能

使用简化后的表达式来定义这个模型，定义为 $Z=A\cdot B\cdot \overline{C}+\overline{A}\cdot B\cdot D$，很明显减少了一个3输入与门，并且或门也变成了2输入，从而减小了面积。

## 16.3 改善性能

这里举一个简单的例子，计算加法$x=a+b+c+d$，所有的变量都是数字。每次用加法器将两个数相加，然后结果作为下一次的输入。对应的数据流程图如图16-4所示。

186

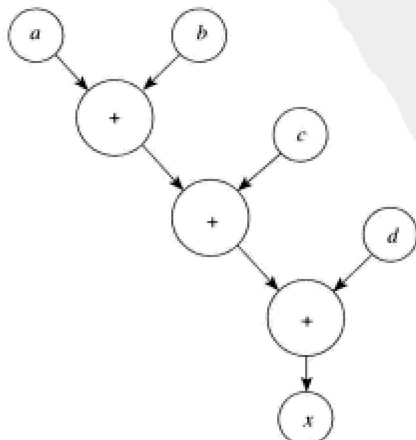



图16-4 加法操作的原始数据流程图

这一实现需要3个加法器，用3个周期才能得到结果。如果我们更系统地使用同样的资源，将可以减少到两个周期，只要采用图16-5所示的结构即可。

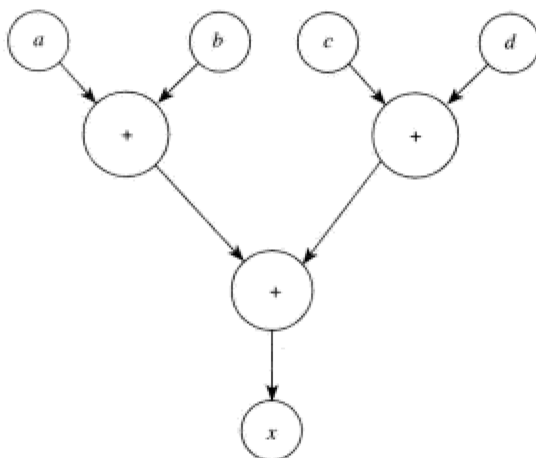



图16-5 精简周期实现

这是一个经典的例子，表达式树简化了，控制路径使用更少的时钟周期，而数据路径的结果完全相同。我们也可以对这个例子加以修改，只使用一个加法器，用寄存器存储中间和，用流水线方式得到最终的结果。这可能会花更长的时间，但是电路面积却是最小的，因为只要一个加法器（当然，代价是用了更多的寄存器）。

## 16.4 关键路径分析

逻辑优化的另一种方法是从时序的角度对设计进行关键路径分析。这一过程通常由综合软件自动完成，例如Synopsys公司著名的Design Compiler软件就会自动生成一个综合后的电路图，其中高亮显示了设计中关键的时序路径，这样设计者就可以将重点放在这些区域进行改进，提高整体设计的性能，如图16-6所示。

187

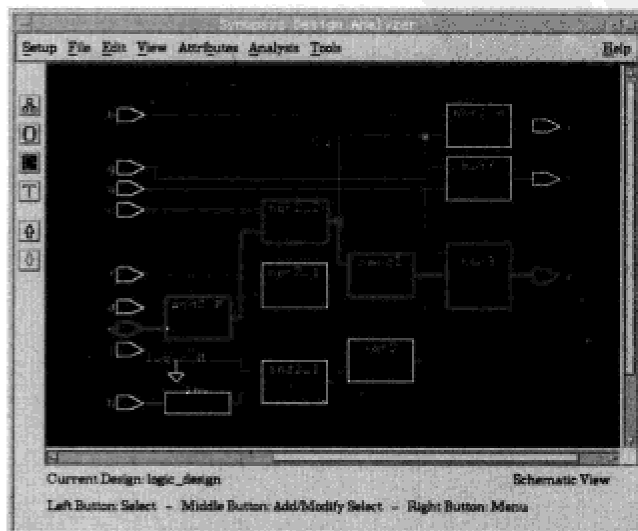



图16-6 关键路径分析

## 16.5 小结

本章讨论了一些改善VHDL FPGA设计性能的技术，并说明了它们是如何工作的。实际的设计优化中，大多数都使用综合软件，但是了解相关的过程还是有用的，这样具体的优化目标就可以在合理的时间内以可控制的方式实现。 188



# 第17章 VHDL-AMS

## 17.1 引言

随着系统集成的层次越来越高，不仅有必要对系统的电子行为进行建模，而且也有必要对系统与“真实世界”的接口以及系统中各元件详细的物理行为进行描述。标准语言（如VHDL-AMS）的出现使其成为可能。现在可以用单一的设计方法描述大量的物理系统并对整个系统进行仿真。在许多应用领域中，这一切正在变得越来越重要。这些领域包括混合信号电子学、电磁接口、集成热建模、电子机械与机械系统［包括微电子机械系统，(MEMS)］、应用流体学（包括水力学和微观应用流体学）以及具有电子控制和各种传感器的电力科学等。

本章介绍如何使用VHDL-AMS语言完成多能量域的行为建模，并举例说明如何在各个域之间进行接口，也提出了对这些学科中各种设计技术的一些看法。本章还描述了基本的框架，说明了如何在标准的包中定义可以为大量模型使用的具有一致性的基础，并给出了具体的例子用以说明这种方法的实际设计细节。例如，对电力系统的集成仿真，包括电特性、磁特性和热效应等。另外也用混合域电子学以及机械系统说明了多能量域行为建模中所涉及的关键概念。

189

用VHDL-AMS对器件建模的基本方法是，定义一个模型实体（entity）和构造体(architecture)。其中模型实体定义了模型与系统之间的接口，包括连接点和参数。同这个实体相关联的许多构造体可以描述模型的行为，例如行为级或者物理级描述。一个完整的模型包括一个单一的实体和一个与之对应的构造体。模型的域或者工艺类型由实体的端口声明中使用的terminal类型来定义。IEEE Std 1076.1.1为多能量域定义了标准的类型，这些能量域包括电力、热学、磁学、机械学和光学系统。在模型的构造体内，每一个能量域类型都有一组定义好的“越过（across）”和“通过（through）”变量（在电子领域内，它们分别是电压和电流），这些可以用来定义模型接口与模型内部行为之间的关系。

在“传统的”电子学领域内，设计VHDL-AMS语言的本意是支持“混合信号”系统(包括数字元件和模拟元件以及它们之间的边界)，主要集中于IC设计方面。然而，VHDL-AMS语言的强大真正开始显现却是在微电子机械系统等多学科领域。在这一章中，作者将重点举几个有趣的例子，说明这种建模方法的优点和多领域的仿真。

## 17.2 VHDL-AMS简介

VHDL-AMS是对标准数字VHDL语言的一种在模拟方面的扩展，可以对混合信号系统进行建模。VHDL-AMS语言于1999年被批准为IEEE Std 1076.1国际标准，不过重要的是，要注意到IEEE 1076.1-1999标准包含了完整的数字VHDL 1076标准，而不是一个子集。

标准并没有为模拟学科指定任何库（例如电气库和机械库等）。这些库的定义是一个单独的任务，由工作组IEEE 1076.1.1完成（于2004年发布）。

为了在上下文中描述这些扩展，需要说明一下VHDL的应用范围，然后将VHDL-AMS放在其旁边，如图17-1所示。

VHDL-AMS语言的扩展包括系统级的转换函数（在拉普拉斯域内，行为级）和电路级微分方程。 190

| 应用级 | 内容 |
|---|---|
| 系统 | Transfer Function |
| 系统 | Specification |
| 芯片 | Algorithms |
| 寄存器 | Truth Tables<br>State Tables |
| 逻辑 | Boolean<br>Equations |
| 电路 | Differential Equations |

VHDL
VHDL-AMS

图17-1 VHDL-AMS语言的应用范围

VHDL-AMS对VHDL的扩展可以总结如下。

(1) 一种新的端口类型TERMINAL，这是基本的模拟引脚。

(2) 一种新的类型NATURE，定义了模拟引脚与变量之间的关系。

(3) 一种新的变量QUANTITY，这是模拟变量。

(4) 一种新的变量赋值类型，用来定义同时解的模拟等式。

(5) 对于时间的微分方程操作符（'DOT）和积分操作符（'INTEG）。

(6) 用于等式的IF语句（IF USE）。

(7) Break语句用于初始化非线性元素。

(8) STEP LIMIT控制，用于限制模拟时间步。

## 17.3 模拟引脚：TERMINAL

为了用VHDL-AMS定义模拟引脚，需要在标准的实体端口声明部分使用关键字TERMINAL。例如，如果一个器件有两个模拟引脚（electrical类型，后面会详细说明），那么实体的基本结构如下所示： 191

```
LIBRARY IEEE;
USE IEEE.ELECTRICAL_SYSTEMS.ALL;
ENTITY model IS
GENERIC();
PORT(
      TERMINAL p : electrical;
      TERMINAL m : electrical
      );
END ENTITY;
```

需要注意的是，由于VHDL-AMS被定义为IEEE标准，因此在使用标准库时，例如electrical类型引脚，需要用到IEEE库中的electrical_systems.all;包。

还要注意，引脚不存在方向的问题。因为模拟引脚是能量保存系统的一部分，所以其方向也同能量保存系统一致。

## 17.4 混合域建模

为了使用标准模型，必须有一个框架来定义标准包中使用的TERMINAL端口和变量。有一个完整的标准IEEE 1076.1.1就是定义标准包的，不过我们先来看一个简化的包（本例是指电气系统包），了解一下这些包是如何组织在一起的。

例如，电气系统模型需要处理以下几个关键的特性：

- 电连接点；
- 电“通过（through）”变量（也就是电流）；
- 电“越过（across）”变量（也就是电压）。

电气系统包需要包含这些元素。

192

首先，需要定义基本的子类型。在所有的模拟系统和类型中，最基本的VHDL类型总是REAL型，所以电压和电流必须被定义为REAL的子类型：

```
Subtype voltage is REAL;
Subtype current is REAL;
```

注意，这两种类型并没有自动单位赋值，而是分别由IEEE 1076.1.1标准中的UNIT和SYMBOL属性处理。例如，对于电压，单位定义为“伏特（Volt）”，符号则定义为“V”。

基本电气类型定义的其余部分分别将这些子类型连接到类型的“通过”和“越过”变量上。

```
PACKAGE electrical_system IS
     SUBTYPE voltage IS real;
     SUBTYPE current IS real;
     NATURE electrical IS
          voltage ACROSS
          current THROUGH
          ground REFERENCE;
END PACKAGE electrical_system;
```

## 17.5 模拟变量：quantity

量（quantity）完全是模拟变量，可以用三种方法定义。自由量（free quantity）是一种简单的模拟变量，与存储能量系统没有关系。分支量（branch quantity）在一个或者多个模拟终端之间有直接的关系。源量（source quantity）用来定义专门的电源功能（例如交流电源和噪声源）。

举例来说，定义一个简单的模拟变量*x*，代表一个电压（但是不直接涉及电气连接），相应的VHDL代码如下：

```
QUANTITY x : voltage;
```

另一方面，两个电气引脚之间的分支有一个“通过”变量（电流）和一个“越过”变量

（电压），这需要一个分支量来定义，所以可以立即得到完整的描述。例如，对于两个引脚*p*和*m*之间的元件电压（*v*）和电流（*i*），可以按以下方式定义完整的量：

```
QUANTITY v across i through p to m;
```

193

## 17.6 VHDL-AMS中的联立方程

在VHDL-AMS中，方程是模拟的，需要同时求解，这与使用逻辑技术并行求解的信号不同，也与顺序求解的变量不同。例如，在VHDL-AMS中，解方程可以使用“==”操作符：

```
Y == x**2;
```

式中的*Y*和*X*必须定义成实数（quantity类型或者其他VHDL变量类型）。

## 17.7 一个VHDL-AMS的例子

### 17.7.1 直流电压源

为了说明前面介绍的一些基本概念，我们举一个直流电压源的例子。这个模型有两个引脚，分别是*p*和*m*，还有一个参数dc_value，用来定义输出电压的值，如图17-2所示。

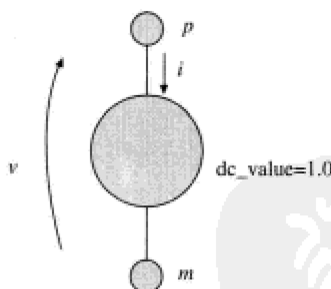



图17-2 基本电压源

用VHDL-AMS建模可以分为两部分，分别是实体和构造体。首先来看实体。因为有两个电气引脚，所以需要使用ieee.electrical_systems.all;这个包，而且端口还要声明为TERMINAL。另外，通用属性dc_value必须定义成实数，其默认值也是实数（例如1.0）：

```
LIBRARY IEEE;
USE IEEE.ELECTRICAL_SYSTEMS.ALL;
ENTITY v_dc IS
GENERIC(
     dc_value : real := 1.0);
PORT(
     TERMINAL p : electrical;
     TERMINAL m : electrical
    );
END ENTITY;
```

194

在构造体中，必须将电压源的电压和电流定义成quantity变量，然后将它们连接到各自的引脚上。电压源的输出等式必须用VHDL-AMS中的模拟等式（也就是“==”操作符）实现功能v = dc_value：

```
ARCHITECTURE simple OF v_dc IS
     QUANTITY v ACROSS I THROUGH p TO m;
BEGIN
     v == dc_value;
END ARCHITECTURE simple;
```

### 17.7.2 电阻

电阻的实体模型与电压源非常相似，也具有两个电气引脚*p*和*m*，以及一个属性，用于表示电阻值，如图17-3所示。

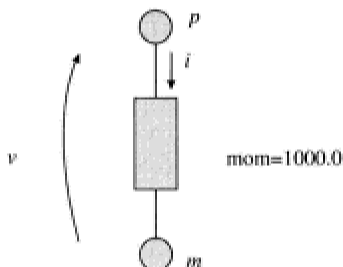



图17-3 VHDL-AMS电阻符号

这一模型可以用VHDL-AMS描述为两部分，分别是实体和构造体。首先看实体。由于有两个电气引脚，所以需要使用ieee.electrical_systems.all;这个包，而且端口还要声明为TERMINAL。另外，通用属性rnom必须定义成实数，其默认值也是实数（例如1000.0）：

```
LIBRARY IEEE;
USE IEEE.ELECTRICAL_SYSTEMS.ALL;
ENTITY resistor IS
GENERIC(
     rnom : real := 1000.0);
PORT(
     TERMINAL p : electrical;
     TERMINAL m : electrical
    );
END ENTITY;
```

195 在构造体中，必须将电阻的电压和电流定义成quantity变量，然后将它们连接到各自的引脚上。电阻的输出等式必须用VHDL-AMS中的模拟等式（也就是“==”操作符）实现功能v = I * rnom：

```
ARCHITECTURE simple OF resistor IS
     QUANTITY v ACROSS I THROUGH p TO m;
BEGIN
     v == I * rnom;
END ARCHITECTURE simple;
```

## 17.8 VHDL-AMS中的微分方程

VHDL-AMS中也可以使用两种微分操作符对线性微分方程进行建模：

(1) 'DOT（求变量对时间的微分）；

(2) 'INTEG（求变量对时间的积分）。

下面用两个例子对其进行说明，一个是电容，一个是电感。首先来看电容的基本表达式：

$$i = C\frac{dV}{dt}$$

使用与电阻类似的模型结构可以定义一个实体和构造体，但是等式怎么办呢？在VHDL-AMS中，操作符'DOT用于电压，表示如下的计算：

```
i == c*v'DOT;
```

因此，完整的VHDL-AMS电容模型可以用下面的代码实现：

```
LIBRARY IEEE;
USE IEEE.ELECTRICAL_SYSTEMS.ALL;
ENTITY capacitor IS
GENERIC(
     cap : real := 1.0e-9);
PORT(
     TERMINAL p : electrical;
     TERMINAL m : electrical
    );
END ENTITY;

ARCHITECTURE simple OF capacitor IS
     QUANTITY v ACROSS I THROUGH p TO m;
BEGIN
     I == cap * v'DOT;
END ARCHITECTURE simple;
```

196

那么电感如何描述呢？电感的基本等式如下：

$$i = \frac{1}{L}\int v\,dt$$

很明显，实现这一等式的最直接方法是使用操作符'INTEG，但是在使用的时候一定要小心。

在使用时，必须考虑初始条件。另外，不同的仿真器也可能出现不同的情况。最简单的实现形式如下：

```
LIBRARY IEEE;
USE IEEE.ELECTRICAL_SYSTEMS.ALL;
ENTITY inductor IS
GENERIC(
     ind : real := 1.0e-9);
PORT(
     TERMINAL p : electrical;
     TERMINAL m : electrical
    );
END ENTITY;
```

```
ARCHITECTURE simple OF inductor IS
     QUANTITY v ACROSS I THROUGH p TO m;
BEGIN
     I == (1.0/ind) * v'INTEG;
END ARCHITECTURE simple;
```

## 17.9 用VHDL-AMS进行混合信号建模

大多数设计工程师比较熟悉“数字”或者“模拟”建模，但是不太理解“混合信号建模”。为了说明“混合信号建模”，有必要先回顾一下模拟建模和数字建模的意义。首先看数字建模的相关技术。

197 数字系统可以用数字逻辑门或者事件来建模。这是一种快速地在结构上仿真数字系统的方法，基于VHDL或者Verilog门级模型。用计算机进行数字系统的仿真常常采用一种基于事件的方法，而不是解微分方程，事件按时间顺序被安排在确定的时间点上，是离散变化的。多事件和连接问题的解决使用逻辑方法实现。数字模型通常是基于门或者逻辑的，最终的仿真结果具有固定的、预定义的电平（例如0和1）。另外，“瞬间”的变化也有可能发生，也就是说状态可以在零上升时间内由0变为1。

相对来说，在模拟领域内，实际电子系统设计的最底层细节是关于模拟仿真器中对模拟方程模型的使用（这种方法的基准是SPICE仿真器）。许多情况下，电路是以网表的形式描述的。所谓网表，就是设计中所有元件的列表，其中包含了各元件的连接点以及参数（例如长度、宽度或者比例），各元件的参数也不完全相同。

每一个器件都使用非线性微分方程建模，这些微分方程必须用Newton-Raphson类型的方法来解。这一方法虽然非常精确，但是也存在不少问题，如下所示。

- 收敛。如果模型不收敛，那么仿真不会产生任何有意义的结果，或者完全无法仿真。
- 振荡。如果有不连续的情况，就无法找到方程的解。
- 时间。可能需要几个小时才能完成仿真，使用详细的器件模型的大型设计甚至需要几天的时间。

在模拟领域，Newton-Raphson方法通常用来寻找方程的解，利用派生值和当前函数值通过迭代得到下一次解。用于非线性方程的基本Newton-Raphson方法定义如下：

$$x_{n+1} = x_n - \frac{F(x_n)}{F'(x_n)}$$

$F(x_n)$和$F'(x_n)$必须是已知的，通过编码放入仿真器（SPICE）中，这是对问题的一种近似解。对198 于VHDL-AMS仿真器而言，必须使用切线法（或者类似的方法）得到派生值，如图17-4所示。

有了这些完全不同的方法后，我们如何将它们放在一起呢？混合信号系统是什么样的呢？在这些例子中，混合了连续的模拟变量和数字事件。这些模型必须有能力描述不同领域之间的边界和转换。检查模拟变量是否越过阈值的基本方法是使用VHDL-AMS中的ABOVE操作符。

例如，若要检查vin的电压是否高于1.0V，可以使用以下的VHDL-AMS代码：

```
if ( vin'above(1.0) ) then
     flag <= true;
end if;
```

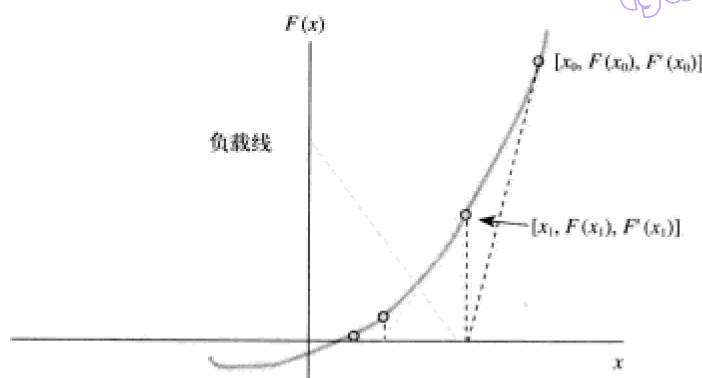



图17-4 Newton–Raphson法

可以进一步将这段代码扩展为参数化，也就是说使用阈值电压参数vth表示阈值电压，该参数在前面定义为通用属性或者常数：

```
if ( vin'above(vth) ) then
    flag <= true;
end if;
```

注意，flag是一个信号，因此可以用在某个过程（process）的敏感列表内，当vin的值越过阈值时，flag信号可以触发使能数字行为的process。如果需要使用相反的条件，也就是电压低于阈值，那么只要使用NOT操作符对条件取反即可：

```
if ( NOT vin'above(vth) ) then
    flag <= true;
end if;
```

数字转换到模拟的接口要比模拟转换到数字的接口更复杂一些，因为输出变量还需要由模拟域控制。 199

当数字域内的事件发生变化时（这很容易用敏感列表检测到），模拟变量必须有正确的值和正确的变化速率。为了实现这一点，可以使用VHDL-AMS中的RAMP属性。

下面看一个简单的例子——数字逻辑到模拟电压的转换接口：

- 当din = “1”时，vout = 5 V；
- 当din = “0”时，vout = 0 V。

可以用以下VHDL-AMS代码实现：

```
process (din) :
begin
    if ( din = '1' ) then
        vdin = 5.0;
    else
        vdin = 0.0;
    end if;
end process;
vout == vdin;
```

很明显，这个简单的接口有些问题，因为vout的转换是没有时间延迟的，这可能会引起

收敛的问题。解决这个问题的方法是在给vout赋值时引入一个RAMP属性，这个属性定义了从一个值连续变化到另一个值所经历的时间：

```
vout == dvin'RAMP(tt)
```

式中的tt（转换时间）要定义为实数（例如tt : real := 1.0e-9;）。

另一种定义转换时间的方法是使用SLEW操作符限制信号的变化斜率。具体做法是在vout的赋值中增加斜率定义，规定信号从一个值连续变化到另一个值所需的时间：

```
vout == dvin'SLEW(max_slew_rate)
```

200

式中的max_slew_rate要定义为实数（例如max_slew_rate : real := 1.0e6;）。

## 17.10 一个基本的开关模型

下面举一个数控开关的例子，该开关具有如下特性：

- ❑ 数字控制输入（*d*）
- ❑ 两个电气引脚（*p*和*m*）
- ❑ 闭合电阻（ron）
- ❑ 断开电阻（roff）
- ❑ 开启时间（ton）
- ❑ 关闭时间（toff）

通过这一简单的概括，就可以使用VHDL-AMS创建一个基本的开关模型。实体如下所示：

```
USE ieee.electrical_system.ALL;
USE ieee.std_logic_1164.ALL;
ENTITY switch IS
    GENERIC ( ron : real := 0.1; — On resistance
           roff : real := 1.0e6; — Off resistance
           ton : real := 1.0e-6; — turn on time
           toff : real := 1.0e-6); — turn off time
    PORT (
    d : IN std_logic;
    TERMINAL p,m : electrical);
END ENTITY switch;
```

构造体的基本结构要求开关上的电压和流过开关的电流要与开关的等效电阻（reff）成比例关系：

```
ARCHITECTURE simple OF switch IS
     QUANTITY v ACROSS i THROUGH p TO m;
     QUANTITY reff : real;
     SIGNAL r_eff : real := roff;
BEGIN
     PROCESS (d)
     BEGIN
          ...
     END;
     i = v / reff;
```

```
END;
```

上述的过程（process）等待输入数字信号*d*的变化，并根据*d*的逻辑值用信号r_eff去对开关的有效电阻（ron或者roff）进行采样。相应的VHDL代码如下： [201]

```
PROCESS (d)
  BEGIN
       if ( d = '1' ) then
             r_eff <= ron;
       else
             r_eff <= roff;
       end if;
  END;
```

当信号r_eff发生变化时，必须使用RAMP函数将其连接到模拟量reff上。前面我们已经说过ramp函数是如何定义上升时间的，其实这个函数也可以定义下降时间。在开关模型构造体中实现这一功能的VHDL-AMS代码如下：

```
reff == r_eff'RAMP ( ton, toff );
i == v / reff;
```

开关模型的完整VHDL-AMS代码为：

```
ARCHITECTURE simple OF switch IS
     QUANTITY v ACROSS i THROUGH p TO m;
     QUANTITY reff : real;
     SIGNAL r_eff : real := roff;
BEGIN
     PROCESS (d)
           BEGIN
                 if ( d = '1' ) then
                       r_eff <= ron;
                 else
                       r_eff <= roff;
                 end if;
           END PROCESS;

           reff == r_eff'RAMP ( ton, toff );
           i == v / reff;
END;
```

## 17.11 基本VHDL-AMS比较器模型

下面再举一个比较器的例子，具有两个输入*p*和*m*，一个接地端gnd和一个数字输出*d*。当输入*p*大于输入*m*时，比较器输出数字1；否则输出0。如图17-5所示。 [202]

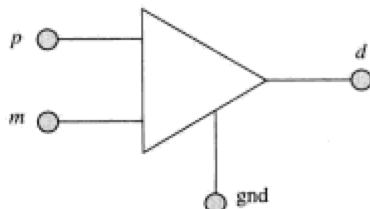



图17-5 比较器

实体中定义了引脚*p*、*m*和gnd，数字输出*d*，输入滞后hys以及传播延迟td：

```
USE ieee.electrical_system.ALL;
USE ieee.std_logic_1164.ALL;
ENTITY comparator IS
    GENERIC (
td : time := 10 ns;
hys : real := 1.0e-6);
    PORT (
d : OUT std_logic := '0';
TERMINAL p,m,gnd : electrical);
END ENTITY comparator;
```

在构造体中，第一步是定义输入电压和基本的处理结构：

```
architecture simple of comparator is
 quantity vin across p to m;
begin
 p1 : process
      constant vh: real := ABS(hys)/2.0;
      constant vl: real := -ABS(hys)/2.0;
 begin
      ...
      wait on vin'above(vh), vin'above(vl);
 end process;
end architecture simple;
```

模拟量vin是输入*p*和*m*之间的电压：

```
quantity vin across p to m;
```

注意，这里并没有定义电流，假定电流为0，所以比较器没有电流流入。另外还要注意，没有定义输入电压偏移量（以后在细化模型时可以增加）。上述过程（process）根据磁滞特性定义了上下阈值vh和vl：

203

```
constant vh : real := ABS(hys)/2.0;
constant vl : real := -ABS(hys)/2.0;
```

随后还有一个等待语句检查vin是否超过这些阈值：

```
wait on vin'above(vh), vin'above(vl);
```

最后的部分是根据vin对阈值的状态增加数字输出逻辑：

```
if vin'above(vh) then
     d <= '1' after td;
elsif not vin'above(vl) then
     d <= '0' after td;
end if;
```

在定义好的延迟时间td后，改变输出*d*的值。

完整的构造体如下所示：

```
architecture simple of comparator is
 quantity vin across p to m;
```

```
begin
 p1 : process
        constant vh : real := ABS(hys)/2.0;
        constant vl : real := -ABS(hys)/2.0;
begin
        if vin'above(vh) then
              d <= '1' after td;
        elsif not vin'above(vl) then
              d <= '0' after td;
        end if;
        wait on vin'above(vh), vin'above(vl);
 end process;
end architecture simple;
```

## 17.12 多领域建模

VHDL-AMS最后一个重要的应用领域是对电子机械系统的建模，尤其是微机械系统（MEMS）。这些器件所用的原理其实是完全一样的，定义机械领域内的模型需要使用机械方程式。值得注意的是，机械模型可以分为转动（角速度和扭矩）和平动（力和距离）类型。混合域系统的一个典型例子是电动机，这里是指直流电动机。采用以下所示的标准电动机方程式，可以看出，参数ke将转子转速与电气域（电动势EMF）联系起来，参数kt则将电流与扭矩联系起来： 204

$$V = L\frac{di}{dt} + iR + Ke\omega$$

$$T = Kti - J\frac{d\omega}{dt} - D\omega$$

上述公式可以用如下的VHDL-AMS模型实现：

```
Library ieee;
use ieee.electrical_systems.all;
use ieee.mechanical_systems.all;

entity dc_motor is
     generic (kt : real;
              j : real;
              r : real;
              ke : real;
              d : real;
              l : real);
     port (terminal p, m : electrical;
           terminal rotor : rotational_v);
end entity dc_motor;

architecture behav of dc_motor is
     quantity w across t through rotor
```

```
    to rotational_v_ref;
        quantity v across i through p to m;
begin
    v == l*i'DOT + i*r + ke*w;
    t == i*kt - j*w'DOT - d*w;
end architecture behav;
```

## 17.13 小结

对于一个综合系统的设计而言，不管是从宏观角度还是从微观角度看，在制造以前就精确地预测出系统的行为变得越来越重要。不管是要保证传感器正确地工作，还是要使综合元件也正确工作，或是分析寄生效应以及非理想效应（例如温度、损耗和非线性等），对于多领域建模的需求从来没有像现在这么强烈。

205 现在，类似于VHDL-AMS的语言为工程师描述这些系统和效应提供了有效的方法，另外，标准化也带来了更多的优点，即增加了互操作性和模型的互换性。对EDA行业的挑战是提供足够的仿真性能，尤其是支持工程设计的建模工具。

摆在FPGA设计者面前的机会是：利用这一建模技术的巨大优点，并用它来保证数字控制器和设计能够有效而稳健地运行于实际的应用中。206

# 第18章　设计优化举例：DES

## 18.1　引言

本书其他章节讨论了设计优化的一些基本内容，但一般是在寄存器传输级（RTL）。虽然也描述了一些行为级建模的内容，但却很少使用行为级综合。本章将研究行为级综合，并将其作为一种创建优化设计的备选方案，而不是使用RTL方法。

本章主要介绍使用MOODS行为综合系统在电子源码书（Electronic Code Book，ECB）模式下设计数据加密标准（DES）内核的实际经验。主要的目标是写出一种高级语言的描述，既有良好的可读性，又可以综合。第二个目标是探测一下单DES和三重DES的面积/延迟设计空间。对综合前（行为级）的设计和综合后（RTL级）的设计都进行了仿真，不但要求两者的输出相同，而且要与测试向量中的期望输出相同。

## 18.2　数据加密标准

数据加密标准（DES）是一种良好的加密算法，于20世纪80年代被美国国家标准与技术协会（National Institute of Standards and Technology，NIST）首次标准化。在本书后面关于安全系统的章节中将详细讨论这一算法，这里只给出此算法的一些基本信息。



目前，DES已经被AES（Advanced Encryption Algorithm）普遍代替，人们都在研究三重加密算法，称为三重DES，或者简写为TDES。这一算法使用相同的DES内核，只不过用不同的密钥计算三次。DES算法很小但速度很快，主要的操作是混合与置换（只有极少的计算量），因此非常适合硬件实现。

## 18.3　MOODS

MOODS（Multiple Objective Optimization in Control and Datapath Synthesis）是英国南安普顿大学开发的一种高级行为综合工具。MOODS以行为级VHDL代码为输入，并将输入转换为功能上与行为级代码相同的结构级VHDL代码。MOODS使用优化和设计空间探测技术得到适当的RTL设计，以满足设计者的约束与要求。

优化器将行为级VHDL代码转换为一种可以用简单的数据流程图（DFG）表示的形式，这种形式可以令控制流程更加容易优化。实际上，这就是一种很容易转换为RTL级VHDL代码的状态机。对于面积的优化，可以通过使用多路器共享数据单元（例如寄存器）的方式来实现；对于时序的优化，可以通过将多个数据单元结合起来减少需要的时钟周期数来实现。

## 18.4　初始设计

### 18.4.1　简介

DES算法的总体结构如图18-1所示。

核心算法每轮使用不同的子密钥重复16次。这些子密钥长度48bit，由原始的56bit密钥产生。使用功能分解形式（例如，创建一些函数，分别代表DES算法中的等价函数）直接将这一算法转换为VHDL代码。

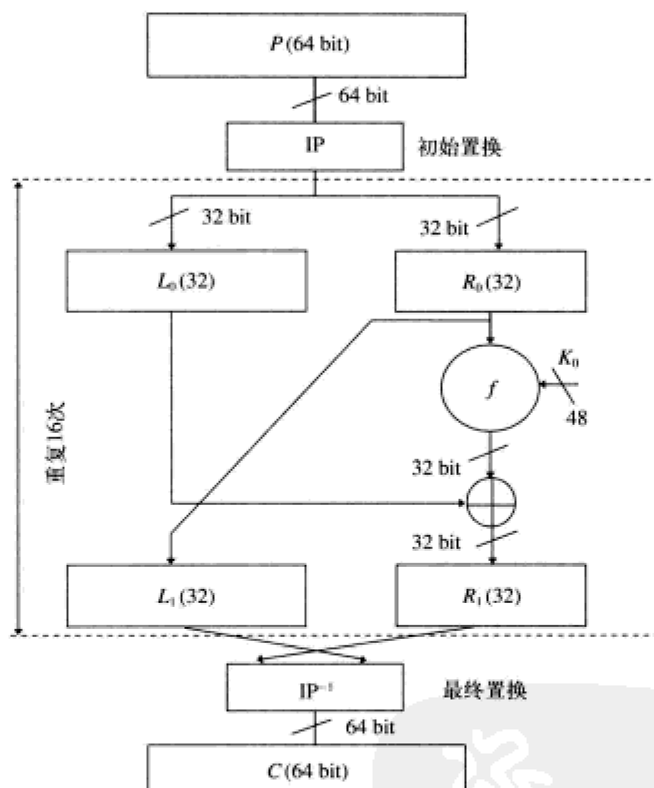



图18-1 DES算法总体结构

## 18.4.2 总体结构

208

本设计的第一阶段是创建一个实体（entity）和一个构造体（architecture），描述必要的输入输出信号，其中有一个描述整体算法的过程（process）。大概的VHDL代码如下所示：

```
library ieee;
use ieee.std_logic_1164.all;
entity DES is
  port (
    plaintext : in std_logic_vector(1 to 64);
    key : in std_logic_vector(1 to 64);
    encrypt : in std_logic;
    go : in std_logic;
    ciphertext : out std_logic_vector(1 to 64);
    done : out std_logic
  );
end;
```

```
architecture behavior of DES is
  subtype vec56 is std_logic_vector(1 to 56);
  ...
  subtype vec64 is std_logic_vector(1 to 64);
begin
  process
  begin
    wait until go = '1';
    done <= '0';
    wait for 0 ns;
    ciphertext <=
      des_core(plaintext, key_reduce(key), encrypt);
    done <= '1';
  end process;
end;
```

209

这一过程是对DES主程序的直接实现。唯一特别的地方在于，模型在开始处理数据前要等待信号go变为1，处理完数据之后将信号done置为1，这是一种基本的握手协议。

算法需要两个函数：key_reduce和des_core。前者将密钥中的奇偶校验位去掉，后者则执行全部的DES算法。函数key_reduce将密钥从64bit减少到56bit，并对某些位进行置换，以便构成子密钥的初始状态。

```
function key_reduce(key : in vec64) return vec56 is
--moods inline
begin
  return
  key(57) & key(49) & key(41) & key(33) &
  ...
  key(28) & key(20) & key(12) & key(4);
end;
```

编译器引导指令--moods inline可以让综合器将这个函数作为内嵌函数。这样，优化器对电路就有了更大的优化空间。函数des_core对一组输入数据进行16次的基本DES运算，每次运算使用不同的子密钥：

```
function des_core
  --moods inline
  (plaintext : vec64;
  key : vec56;
  encrypt : std_logic)
return vec64
is
    variable data : vec64;
    variable working_key : vec56 := key;
begin
  data := initial_permutation(plaintext);
```

```
  for round in 0 to 15 loop
    working_key :=
      key_rotate(working_key,round,encrypt);
    data := data(33 to 64) &
      (f(data(33 to 64),key_compress(working_key))
    xor
    data(1 to 32));
  end loop;
  return
    final_permutation(data(33 to 64) & data(1 to 32));
end;
```

DES算法由密钥转换函数key_rotate和key_compress以及数据转换函数initial_ permutation、f和final_permutation构成。

210

## 18.4.3 数据转换

数据转换函数initial_permutation和final_permutation就是使用一系列关联事件的简单的硬连线方式的位置换：

```
function initial_permutation(data : vec64) return vec64 is
  --moods inline
begin
  return
    data(58) & data(50) & data(42) & data(34) &
    ...
    data(31) & data(23) & data(15) & data(7);
end;

function final_permutation(data : in vec64) return vec64 is
  --moods inline
begin
  return
    data(40) & data(8) & data(48) & data(16) &
    ...
    data(49) & data(17) & data(57) & data(25);
end;
```

函数f是主要的数据转换函数，对输入数据的右边32bit进行16次操作。该函数的第二个参数是key_compress函数产生的48bit子密钥：

```
function f(data : vec32; subkey : vec48) return vec32 is
  --moods inline
begin
  return permute(substitute(expand(data) xor
  subkey));
end;
```

函数f首先输入32bit数据，然后使用扩展算法将32bit输入扩展为48bit输出。这里的扩展

算法也是一种位的重新排序，将输入数据以特定的方式进行复制，从而把32bit数据扩展为48bit：

```
function expand(data : vec32) return vec48 is
  --moods inline
begin
  return
    data(32) & data(1) & data(2) &
    ...
    data(31) & data(32) & data(1);
end;
```

211

扩展后的数据字与子密钥按位异或后进入置换模块。置换模块将输入数据中每6bit置换为4bit（记住，原始输入已经从32bit扩展为48bit，因而共有8个置换）。置换模块还有一个作用就是将输出数据减少到32bit。置换的具体算法为：首先将48bit输入分为8组，每组6bit。然后用这6bit数据查找一个6输入的置换表，该置换表称为S盒。在初始实现中，我们用一个只读存储器（ROM）来存储所有的置换模式。置换模块将块索引和输入数据结合起来构成一个地址，用来查找S盒中相应的置换值。地址的计算由函数smap执行：

```
function smap(index : vec3; data : vec6) return vec4 is
  --moods inline
  type S_block_type is
    array (0 to 511) of natural range 0 to 15;
  constant S_block : S_block_type :=
    --moods ROM
  (
  14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12, 5, 9, 0, 7,
  ...
  2, 1, 14, 7, 4, 10, 8, 13, 15, 12, 9, 0, 3, 5, 6, 11
  );
begin
  return
    vec4(to_unsigned(S_block(to_integer(unsigned(
index & data(1) & data(6) & data(2 to 5)))),4));
end;
```

在置换函数中，分别调用8次smap函数执行8次置换：

```
function substitute(data : vec48) return vec32 is
  --moods inline
begin
  return
    smap("000",data(1 to 6)) &
    ...
    smap("111",data(43 to 48));
end;
```

数据置换中最后一个阶段是另一个位置换操作：

```
function permute (data : in vec32) return vec32 is
  --moods inline
```

212

```
begin
  return
    data(16) & data(7) & data(20) & data(21) &
    ...
    data(22) & data(11) & data(4) & data(25);
end;
```

以上所描述的函数定义了算法中的全部数据路径。

### 18.4.4 密钥转换

加密密钥也需要多次转换，在每次数据转换前，密钥要进行循环移位，然后从56bit密钥中提取48bit的子密钥。循环移位是密钥转换中最复杂的部分。首先，56bit密钥分成两半，每一半根据DES算法正在执行的轮数分别循环移位0、1或者2bit。循环移位的方向是：加密时循环左移，解密时循环右移。密钥转换的算法分为两个函数，分别是do_rotate和key_rotate。从名称中就可以猜测出来，do_rotate执行循环移位，key_rotate调用do_rotate函数两次，各用于密钥的两个部分。do_rotate函数使用ROM存储每一轮循环移位的距离，轮数从0到15：

```
function do_rotate
  --moods inline
  (key : in vec28;
  round : natural range 0 to 15;
  encrypt : std_logic)
return vec28 is
  type distance_type is
  array (natural range 0 to 15) of integer range 0 to 2;
  constant encrypt_shift_distance : distance_type :=
  -- moods ROM
  (1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1);
  constant decrypt_shift_distance : distance_type :=
  -- moods ROM
  (0, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1);
  variable result : vec28;
begin
  if encrypt = '1' then
  result :=
  vec28(unsigned(key) rol
  encrypt_shift_distance(round));
  else
  result :=
  vec28(unsigned(key) ror
  decrypt_shift_distance(round));
  end if;
  return result;
end;
```

213

key_rotate函数只是简单地调用了两次上面的函数do_rotate：

```
function key_rotate
  --moods inline
  (key : in vec56;
  round : natural range 0 to 15;
  encrypt : std_logic)
  return vec56 is
begin
  return do_rotate(key(1 to 28),round,encrypt) &
    do_rotate(key(29 to 56),round,encrypt);
end;
```

最后，密钥压缩函数key_compress从56bit密钥中选择48bit，并传递给S盒：

```
function key_compress(key : in vec56) return vec48 is
  --moods inline
begin
return
key(14) & key(17) & key(11) & key(24) &
...
key(50) & key(36) & key(29) & key(32);
end;
```

## 18.5 初始综合

使用MOODS对设计进行综合，时序优先级最高，面积优先级次之。目标工艺为Xilinx公司的Virtex库。图18-2显示了综合后设计的控制状态机。所有的状态序列代表处理的过程，这是一个从最后的状态c11到第一个状态c1的不断循环的过程。

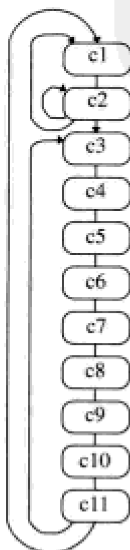



图18-2 初始综合后的控制状态机

前两个状态c1和c2实现的是输入的握手，即信号go触发处理的过程。DES内核的功能则由剩余的状态c3到c11来实现，这是一个从状态c11到c3的循环过程，共循环16次。在这个内

部循环中，共有9个状态，完成整个算法总共需要146个时钟周期，其中包括输入握手需要的2个和DES内核需要的144个。但是，从图18-1所示的原始结构观察，对内部循环进行优化的合理目标是每轮两个周期，乐观的目标是一个周期。很明显，这个设计存在问题。MOODS预测出的面积和时序特性列在本章第一个表中标记为（1）的一行，读者可以参考。

## 18.6 优化数据路径

检查主循环中的9个控制状态，并把它们与数据流程图中的控制路径联系起来，我们将发现最后8个周期执行S盒的功能，前2个周期主要是进行密钥转换。第二个状态是一个重叠的状态，既有密钥的转换，又有数据的转换。最后8个周期的问题是不言自明的，因为有8次置换，而它们又分别由8个控制状态执行。很明显，这样一来，每次置换都在独立的控制状态中，从而使优化很难进行。不难发现，这些状态中，每一个包含的只是一些寄存器赋值、数据拼接和读取ROM操作。最后一个问题，ROM是同步电路，所以S盒的ROM每个时钟周期只能被访问一次，换句话说，就是每个控制状态访问一次。正是由于这一点，数据路径的操作无法并行执行。

解决这个问题已经超越了行为综合的能力范围，因为这需要在比自动提取更高的层次上理解数据流程。所以，为了解决问题，必须对原始设计进行修改。

有两个比较明显的方案可以解决这个问题，一个是将S盒分成8个更小的ROM，以便并行访问，另一个是用非ROM方式实现S盒，将每次读取用一个译码器实现，总共8个译码器。后一种方案看起来更简单，但是结果将产生8个512路的译码器，这将是一个非常大的电路。因此，将S盒分成8个更小的ROM才是一个实用的方案。将置换函数重写，使用8个小ROM，代码如下：

```
function substitute(data : vec48) return vec32 is
  --moods inline
  type S_block_type is
    array 0 to 63) of natural range 0 to 15;
  constant S_block0 : S_block_type := ( ... );
  --moods ROM
  ...
  constant S_block7 : S_block_type := ( ... );
  --moods ROM
begin
  return std_logic_vector(to_unsigned(S_block0(to_integer(
    unsigned(data(1) & data(6) & data(2 to 5)))),4)) &
    ...
    std_logic_vector(to_unsigned(S_block7(to_integer(
    unsigned(data(43) & data(48) & data(44 to 47)))),4));
end;
```

对上述代码重新综合，结果如图18-3所示。内部的循环已经减少到了两个状态，检查最后一个状态会发现，所有的S盒置换操作都发生在状态c4中。密钥转换则发生在状态c3和c4中。

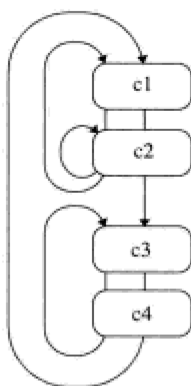



图18-3 优化S盒后的控制状态机

这次优化的一个有趣的副作用是综合出了一个更小的设计。MOODS预测出的面积和时序特性列在18.8节的表中标记为（2）的一行。

## 优化密钥转换

检查主循环中的两个状态，都涉及密钥转换，这两个状态执行ROM读取和循环移位操作。检查原始的key_rotate函数会发现，每次调用都会从ROM中读取两次移位距离，这和前面讨论的S盒ROM存在同样的问题。因为ROM是同步的，每个时钟周期只能访问一次，所以循环移位操作至少需要两个周期。为了解决这个问题，将这个函数重写，每次调用只读取一次ROM：

```
if encrypt = '1' then
  distance := encrypt_shift_distance(round);
  result :=
    vec28(unsigned(key(1 to 28)) rol distance) &
    vec28(unsigned(key(29 to 56)) rol distance);
else
  distance := decrypt_shift_distance(round);
  result :=
    vec28(unsigned(key(1 to 28)) ror distance) &
    vec28(unsigned(key(29 to 56)) ror distance);
end if;
```

重新综合后得到新的控制状态图，如图18-4所示。内部循环减少到了一个状态c3，既包 217

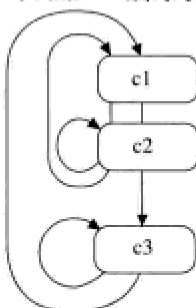



图18-4 优化密钥循环移位后的控制状态机

含密钥转换，又包含数据转换，运算过程重复16次。状态c1和c2实现输入握手。

这样，本次优化就达到了内核运算每轮使用一个时钟周期的优化目标。MOODS预测出的面积和时序特性列在18.8节的表中标记为（3）的一行。

## 18.7 最终综合

众所周知，换一种思路考察循环移位算法，可以将函数key_rotate进一步简化。例如，循环右移1bit可以用循环左移27bit代替（数据为28bit）。这样就可以去掉一条条件语句，该条件语句有可能对优化造成一定负面影响。这就意味着，对于加密和解密没有必要使用不同的循环移位算法。重写后的代码如下：

```
function key_rotate
  --moods inline
  (key : vec56;
  round : natural range 0 to 15;
  encrypt : std_logic)
return vec56 is
  type distance_type is
  array (natural range 0 to 31) of integer range
  0 to 31;
constant shift_distance : distance_type :=
  --moods ROM
  (0, 1, 2, 2, 2, 2, 2, 2,
  1, 2, 2, 2, 2, 2, 2, 1,
  27, 27, 26, 26, 26, 26, 26, 26,
  27, 26, 26, 26, 26, 26, 26, 27);
  variable distance : natural range 0 to 31;
begin
  distance := shift_distance(to_integer(unsigned(
    encrypt & to_unsigned(round,4))));
  return vec28(unsigned(key(1 to 28)) ror distance) &
    vec28(unsigned(key(29 to 56)) ror distance);
end;
```

这一设计的状态机与图18-4所示的设计状态机基本相同。可以看出，这一版本的设计比前一版本的稍微慢了些，但是面积却小了很多。

218

MOODS预测出的面积和时序特性列在18.8 节的表中标记为（4）的一行。

## 18.8 结果

前面讨论过的各种版本DES设计，MOODS预测的结果总结如下表所示。

单DES设计的物理度量

| 设　　计 | 面积（片） | 延迟（周） | 时钟（ns） | 吞吐量（MB/s） |
|---|---|---|---|---|
| （1）初始设计 | 552 | 146 | 7.8 | 7.12 |
| （2）优化S盒后的设计 | 426 | 34 | 7.1 | 35.2 |
| （3）优化密钥转换的设计 | 489 | 18 | 7.1 | 62.6 |
| （4）优化条件分支的设计 | 307 | 18 | 8.4 | 52.9 |

从表中可以看出，设计（3）速度最快，而设计（4）面积最小。图18-5给出了4个设计的面积与吞吐性能对比。横轴代表面积，纵轴代表吞吐性能。

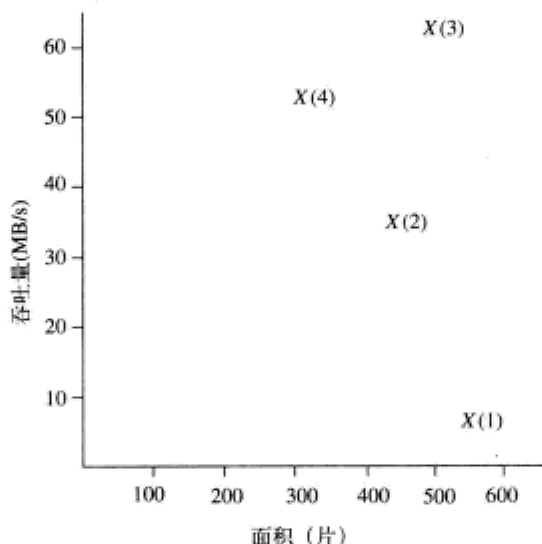



图18-5 全部DES设计的面积与吞吐性能对比

## 18.9 三重DES

### 18.9.1 引言

219

上面开发的DES内核也可以作为三重DES的内核使用。

三重DES就是进行三次加密。三重DES的共同形式称为EDE2，即加密（encrypt）、解密（decrypt）、再加密（encrypt again），但是使用两组不同的密钥，第一组密钥用于两次加密，第二组密钥用于解密。显然，在这个设计中，可能存在多种不同的实现方法。下面一节详细讨论了这些方法。在所有方法中，都使用面积最小的DES实现［设计（4）］作为DES内核。

### 18.9.2 面积最小：迭代实现

为了获得面积最小的实现，3个处理阶段只使用一个DES内核。数据3次通过这个DES内核，但是每次的密钥和加解密模式不同，由此而实现EDE2算法。下面使用了两种不同类型的VHDL编码，主要差异在于每个加密步骤选择输入的方法不同。第一种类型使用了case语句，第二种类型则使用数组。使用case语句的VHDL设计如下所示：

```
library ieee;
use ieee.std_logic_1164.all;
entity tdes_ede2_iterative is
  port(
    plaintext : in std_logic_vector(1 to 64);
    key1 : in std_logic_vector(1 to 64);
    key2 : in std_logic_vector(1 to 64);
```

```
    encrypt : in std_logic;
    go : in std_logic;
    ciphertext : out std_logic_vector(1 to 64);
    done : out std_logic);
end;
architecture behavior of tdes_ede2_iterative is
  ...
begin
  process
    variable data : vec64;
    variable key : vec56;
    variable mode : std_logic;
  begin
    wait until go = '1';
    done <= '0';
    wait for 0 ns;
    data := plaintext;
  for i in 0 to 2 loop
    case i is
    when 1=>
      key := key_reduce(key2);
      mode := not encrypt;
    when others =>
        key := key_reduce(key1);
        mode := encrypt;
      end case;
      data := des_core(data,key,mode);
    end loop;
    ciphertext <= data;
    done <= '1';
  end process;
end;
```

220

可以看出，这里使用case语句为每次迭代选择适当的密钥和加密模式。这种方式综合后的一些特征列在18.10节给出的表中。

核心DES算法需要48个周期（3次迭代，每次迭代需要16轮，每轮一个周期），另外还需要3个周期，这是由于每次迭代中，case语句选择密钥需要一个周期，3次迭代需要3个周期。第二种类型使用数组来存储密钥和模式，然后每次迭代使用不同的索引访问这些数组即可。处理的过程如下：

```
process
  ...
  type keys_type is array (0 to 2) of vec56;
  variable keys : keys_type;
  type modes_type is array (0 to 2) of std_logic;
  variable modes : modes_type;
begin
```

```
...
modes := (encrypt, not encrypt, encrypt);
keys := (key_reduce(key1),
  key_reduce(key2),
  key_reduce(key1));
for i in 0 to 2 loop
  data := des_core(data,keys(i),modes(i));
end loop;
...
```

从中我们发现，这种方式的延迟与case语句的情况相同，但是面积却增大了约25%。这主要是由于使用了寄存器数组，导致多用了大约200个触发器。显然，这两种方式中，case语句的设计是效率最高的，因此我们放弃数组方式而使用case方式。

221

### 18.9.3 延迟最小：流水线方式

为了使设计的延迟最小，使用3个DES内核构成流水线结构。这样，每18个周期就可以输入一次数据，这只是单个DES内核的延迟，不过由于流水线长度的原因，得到第一个运算结果却需要50个时钟周期。这一电路复制了3个DES过程：

```
architecture behavior of tdes_ede2_pipe is
  ...
  signal intermediate1,intermediate2 : vec64;
begin
  process
  begin
    wait until go = '1';
    intermediate1 <=
      des_core(plaintext,key_reduce(key1),encrypt);
  end process;
  process
  begin
    wait until go = '1';
    intermediate2 <=
      des_core(intermediate1,key_reduce(key2),not
      encrypt);
  end process;
  process
  begin
    wait until go = '1';
    done <= '0';
    wait for 0 ns;
    ciphertext <=
      des_core(intermediate2,key_reduce(key1),
      encrypt);
    done <= '1';
  end process;
end;
```

需要注意的是，done信号仅仅由其中一个内核驱动，假如3个内核综合为同样的时序延迟，这将给出正确的结果，实际应用中也是如此。这样的设计方法可以使3个内核间不再需要握手信号。MOODS预测出的面积和时序特性列在18.10节的表中。图18-6中的状态机显示了3个独立的过程。例如，第一个过程用状态c2、c3和c4表示。其中前两个状态处理握手，222 状态c4则处理DES内核的16次循环。状态c7是第二个DES内核处理状态，状态c10是第三个。

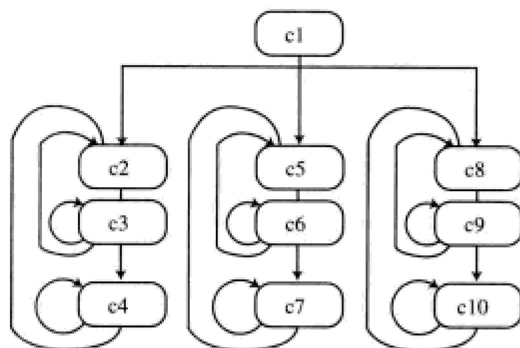



图18-6 流水线方式三重DES的控制状态机

## 18.10 方案比较

前一节的物理度量是由MOODS给出的预测值。为了得到更精确的估计，必须进一步将MOODS输出的结构化VHDL代码进行RTL综合。这一步可以使用Mentor Graphics公司的RTL综合软件Leonardo Spectrum来做。综合结果再由Xilinx公司的ISE Foundation软件进一步做布局布线处理。3种工具（MOODS、Leonardo和Foundation）对3种实现方案（DES、迭代TDES和流水线TDES）的预测结果如下表所示。所有情况下，RTL综合过程中使用厂商提供223 的默认优化设置（同时考虑了面积和时序优化）进行优化处理，选用了最大优化选项。布局布线则选用一个无法达到的时钟周期来执行，以便让Foundation软件给出最快的设计。

| 设　　计 | 工　　具 | 面积（片） | 延迟（周） | 时钟（ns） | 吞吐量（MB/s） |
|---|---|---|---|---|---|
| DES | MOODS | 307 | 18 | 8.4 | 52.9 |
| | Leonardo | 258 | | 13.4 | 33.2 |
| | Foundation | 274 | | 18.4 | 24.2 |
| 迭代TDES | MOODS | 500 | 53 | 8.4 | 18.0 |
| | Leonardo | 381 | | 13.7 | 11.0 |
| | Foundation | 422 | | 17.8 | 8.5 |
| 流水线TDES | MOODS | 920 | 18 | 8.4 | 52.9 |
| | Leonardo | 774 | | 13.7 | 32.4 |
| | Foundation | 826 | | 18.4 | 24.2 |

表中的结果显示，MOODS倾向于过高估计面积，而低估了时序。这都是预料之中的结果。过高估计面积是因为MOODS工作于行为级，不可能预测出逻辑最小化的效果。而过低估计时序延迟则是因为不能预测出路径延迟。

## 18.11 小结

本章的讨论说明，使用高抽象级的VHDL代码设计复杂的算法（例如DES）并得到可综合的设计是完全有可能的。但是，综合的过程不是、也不可能完全自动化，要优化设计结构，使综合工具给出最优化的设计，人工的引导仍然是必要的。尽管修改发生在高层的设计中，但是最终的设计仍然是可读和抽象的。没有必要下降到低层VHDL来实现DES。三重DES的实现表明，在高抽象层上写的VHDL代码很容易实现重用。在4个工作日内完成DES算法的实现，完成两种三重DES算法的实现（包括测试），这是相当好的成绩，这也证明了行为级综合工具的能力。

224



# 第五部分　基本技术

第五部分的主要目的是为读者提供一组非常基础的VHDL标准功能。这主要是针对那些刚接触VHDL的读者，他们可能需要了解VHDL中一些非常简单的功能。本部分内容描述了一些标准的技术，用来实现寄存器、计数器、译码器、多路复用器、锁存器和触发器等，另外还包括一些背景知识，例如定点算术操作、二进制乘法、有限状态机、串并转换和并串转换以及ALU（算术逻辑单元）功能。

书中给出的VHDL代码都是用来举例说明的，因而与前文介绍效率、速度和面积时所用的代码相比更加清晰、简单，若要实际进行物理实现还需要优化和进一步设计。设计VHDL代码的目的是让设计者理解这些操作是如何工作的，让设计者以少量相关知识就可以实现自己的设计功能。

# 第19章 计 数 器

## 19.1 引言

在实际的电子系统中，触发器最普遍的应用之一是计数器。微处理器使用计数器对程序指令进行计数（称为程序计数器，PC），用于访问存储器（例如ROM）中连续的地址或者检查测试的进度。计数器可以从任意值开始计数，不过绝大多数情况下都是从0开始，既可以递增计数也可以递减计数。计数器每次变化的值也可能大于1，也可能以不同的顺序计数（例如格雷码计数器、二进制码计数器或者BCD码计数器）。

## 19.2 基本二进制计数器

在许多情况下，实现起来最简单的计数器是二进制计数器。其基本结构是一组触发器（一个寄存器），受复位端控制（将计数器复位为0），时钟信号使计数器递增。最后是计数器输出，其宽度由通用参数$n$决定，也就是决定计数器的大小。计数器的符号如图19-1所示。注意，复位信号是低有效，按照IEEE标准格式，计数器符号中将计数器和复位信号画在不同的块中。

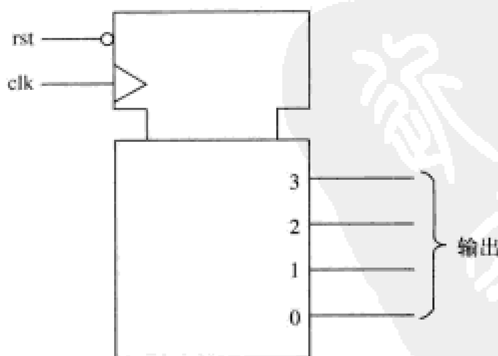



图19-1 简单二进制计数器

227

从FPGA实现的角度看，通用参数$n$的值也定义了需要的D触发器的个数（通常是一个查找表），因此也就决定了FPGA中资源的使用量。这样一个计数器的简单实现如下列代码所示：

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity counter is
  generic (
```

```
        n : integer := 4);
    port (
        clk : in std_logic;
        rst : in std_logic;
        output : out std_logic_vector((n-1) downto 0)
    );
end;

architecture simple of counter is
begin
  process(clk,rst)
    variable count : unsigned((n-1) downto 0);
  begin
    if rst= '0' then
      count := (others => '0');
    elsif rising_edge(clk) then
      count := count + 1;
    end if;
    output <= std_logic_vector(count);
  end process;
end;
```

这个实现方法中有一点很重要：这里的计数器实际上是一个状态机，但是我们并没有显式地定义下一个状态的产生逻辑，这将由综合软件处理。现在，可以用一个简单的测试平台测试这个计数器。首先对计数器复位，然后为计数器提供时钟，使计数器一轮接一轮地计数。测试平台代码如下： 228

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity CounterTest is
end CounterTest;

architecture stimulus of CounterTest is
      signal rst : std_logic := '0';
      signal clk : std_logic := '0';
      signal count : std_logic_vector (3 downto 0);

      component counter
          port(
              clk : in std_logic;
              rst : in std_logic;
              output : out std_logic_vector(3 downto 0)
          );
  end component;
  for all : counter use entity work.counter ;

begin
```

```
    CUT : counter port map(clk=>clk,rst=>rst,
      output=>count);
    clk <= not clk after 1 us;
    process
    begin
    rst <= '0','1' after 2.5 us;
    wait;
    end process;
end;
```

在这个简单的测试平台中，对计数器的复位时间为2.5μs，然后计数器将在时钟上升沿开始计数（时钟频率为500kHz）。

如果对这个计数器模型进行分解，就会发现几个有意思的现象。首先是需要定义一个内部变量count，而不能直接对输出变量$q$进行递增操作。输出变量$q$已经被定义成标准逻辑向量std_logic_vector，并且作为输出，因此不能将它用作等式的输入变量。因此需要定义一个局部变量来存储计数器的当前值。

229

首先要决定的是应该用变量还是信号。在这个例子中，需要使用内部变量。我们可以把它用作一个时序信号，它的值是立即改变的，所以要用变量。如果我们选用了信号，那么它的值只在周期结束时（也就是过程下一次激活时）才改变。

第二个决定是计数器变量应该用什么类型。输出变量是std_logic_vector类型，该类型比std_logic类型的数组好，我们就不用单独指定一个字中的各个位，这都是自动完成的。但是，主要的缺点是std_logic_vector类型不支持简单的算术操作，例如加法。在这个例子中，我们想让计数器的VHDL定义尽量简单，所以最好的折中方案是使用有符号或无符号类型，它们既有位操作定义，又有算术操作功能。另外，我们还希望std_logic_vector的各位与计数器的各位直接映射在一起，所以使用无符号类型最合适。这样，内部计数器变量count的声明就是如下形式：

```
variable count : unsigned((n-1) downto 0);
```

模型的最后一部分是将内部变量count的值赋给外部std_logic_vector输出$q$。从无符号类型向std_logic_vector类型的转换很简单，只要使用标准的类型转换即可：

```
q <= std_logic_vector(count);
```

由于$q$和count的基本类型是一致的，因此这种转换可以直接完成。

## 19.3　综合简单的二进制计数器

当对这段VHDL代码进行综合时会发生什么情况呢？为测试这一点，我们将这个简单的二进制计数器的VHDL代码输入到一个典型的RTL综合软件包（Leonardo Spectrum）中，得到综合后的VHDL代码如下：

230

```
entity counter is
    port (
      clk : IN std_logic;
      rst : IN std_logic;
      output : OUT std_logic_vector (3 DOWNTO 0));
end counter;
```

```
architecture simple of counter is
    signal clk_int, rst_int, output_dup0_3,
      output_dup0_2, output_dup0_1,
      output_dup0_0, output_nx4, output_nx7,
      output_nx10, NOT_rst,
      output_NOT_a_0: std_logic;

begin
    output_obuf_0 : OBUF port map (O=>output(0),
      I=>output_dup0_0);
    output_obuf_1 : OBUF port map (O=>output(1),
      I=>output_dup0_1);
    output_obuf_2 : OBUF port map (O=>output(2),
      I=>output_dup0_2);
    output_obuf_3 : OBUF port map (O=>output(3),
      I=>output_dup0_3);
    rst_ibuf : IBUF port map (O=>rst_int, I=>rst);
    output_3_EXMPLR_EXMPLR : FDC port map
      (Q=>output_dup0_3, D=>output_nx4,
      C=>clk_int, CLR=>NOT_rst);
    output_2_EXMPLR_EXMPLR : FDC port map
      (Q=>output_dup0_2, D=>output_nx7,
      C=>clk_int, CLR=>NOT_rst);
    output_1_EXMPLR_EXMPLR : FDC port map
      (Q=>output_dup0_1, D=>output_nx10,
      C=>clk_int, CLR=>NOT_rst);
    output_0_EXMPLR_EXMPLR : FDC port map
      (Q=>output_dup0_0, D=> output_NOT_a_0,
      C=>clk_int, CLR=>NOT_rst);
    clk_ibuf : BUFGP port map ( O=>clk_int, I=>clk);
    output_nx4 <= (not output_dup0_3 and output_dup0_2
      and output_dup0_1 and output_dup0_0) or
      (output_dup0_3 and not output_dup0_0) or
      (output_dup0_3 and not output_dup0_2) or
      (output_dup0_3 and not output_dup0_1);
    output_nx7 <= (output_dup0_2 and not output_dup0_0)
      or (not output_dup0_2 and output_dup0_1 and
      output_dup0_0) or (output_dup0_2 and not
      output_dup0_1);
    output_nx10 <= (output_dup0_0 and not
      output_dup0_1) or (not output_dup0_0 and
      output_dup0_1);
    NOT_rst <= (not rst_int);
    output_NOT_a_0 <= (not output_dup0_0);
end simple;
```

这个模型第一个明显的特征是，它比原来的RTL代码长很多。第二个特征也很明显，综

231 合后，必须定义物理门电路逻辑。最后，输出做了缓冲，因而在最终的模型中多了一些门电路。打开优化报告看一下，对FPGA中资源使用情况（器件为Xilinx Virtex-II Pro）的统计信息如下：

```
Cell      Library  References      Total Area
==========================================================
BUFGP     xcv2p    1 x         1   1 BUFGP
FDC       xcv2p    4 x         1   4 Dffs or Latches
IBUF      xcv2p    1 x         1   1 IBUF
LUT1      xcv2p    2 x         1   2 Function Generators
LUT2      xcv2p    1 x         1   1 Function Generators
LUT3      xcv2p    1 x         1   1 Function Generators
LUT4      xcv2p    1 x         1   1 Function Generators
OBUF      xcv2p    4 x         1   4 OBUF

Number of ports :                                        6
  Number of nets :                                      17
  Number of instances :                                 15
  Number of references to this view :                    0
Total accumulated area :
  Number of BUFGP :                                      1
  Number of Dffs or Latches :                            4
  Number of Function Generators :                        5
  Number of IBUF :                                       1
  Number of OBUF :                                       4
  Number of gates :                                      5
  Number of accumulated instances :                     15
Number of global buffers used:                           1
**********************************************************
Device Utilization for 2VP2fg256
**********************************************************
Resource                    Used    Avail     Utilization
----------------------------------------------------------
IOs                         5       140       3.57%
Global Buffers              1       16        6.25%
Function Generators         5       2816      0.18%
CLB Slices                  3       1408      0.21%
Dffs or Latches             4       3236      0.12%
Block RAMs                  0       12        0.00%
Block Multipliers           0       12        0.00%
```

从这个简单的例子中可以看出，FPGA的总体利用率虽然很低，但是却分配了各种相关的资源，包括IO、缓存和功能模块。对于FPGA设计而言，这一点很重要，因为即使整个器件的资源没有被充分利用，但是某种特殊的资源（例如IO）却有可能已经被用完。综合后的VHDL代码再输入到物理布局布线软件工具（例如Xilinx Design Navigator）中，产生最后的

232 位流文件，以便下载到器件中工作。

## 19.4 移位寄存器

严格来说，移位寄存器并不是计数器，不过将其看作计数器却有助于理解其他计数器的原理，因为移位寄存器经过很小的改变就可以转换为计数器。本书中我们将比较详细地分析一下这种器件，不过只考虑简单的情况，即移位寄存器只有1bit宽度，输入数据存储在寄存器的最低位，每个时钟沿向高位移动一位。如果使用n比特的寄存器，并且给出时钟沿前后的状态，那么移位寄存器的功能就很清楚了，如图19-2所示。

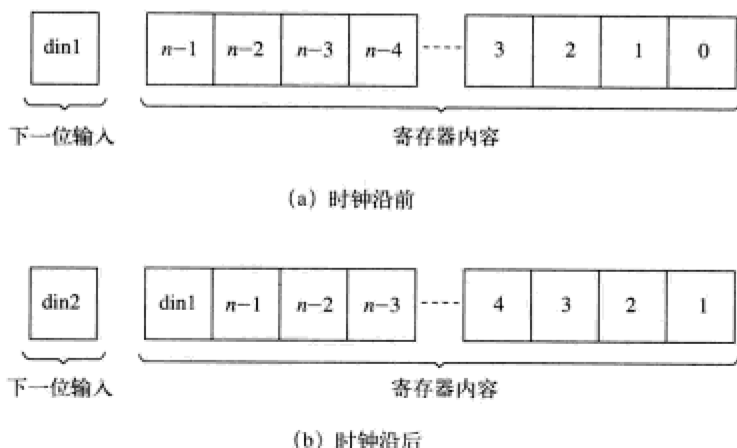



图19-2 移位寄存器功能

基本的移位寄存器可以用以下VHDL代码实现：

```
library ieee;
use ieee.std_logic_1164.all;

entity shift_register is
  generic (
      n : integer := 4);
  port (
        clk : in std_logic;
        rst : in std_logic;
        din : in std_logic;
        q : out std_logic_vector((n-1) downto 0)
  );
end entity;
architecture simple of shift_register is
begin
  process(clk,rst)
    variable shift_reg : std_logic_vector((n-1) downto 0);
  begin
    if rst= '0' then
      shift_reg := (others => '0');
    elsif rising_edge(clk)  then
```

```
      shift_reg := shift_reg(n-2 downto 0) & din;
    end if;
    q <= shift_reg;
  end process;
end architecture simple;
```

有意思的是，这个模型与简单二进制计数器非常相似，只有一些细微的差异。在模型中，我们定义了内部变量shift_reg，不过和计数器不同，这个变量不需要执行算术功能，这样我们就不用将它定义为无符号类型，而是直接定义为std_logic_vector类型（与输出$q$相同）。

注意，这里我们使用了异步复位。前面已经讨论过完全同步的置位和复位，因此如果必要的话也可以使用。

计数器和移位寄存器之间的基本差异是位的移位方式。在计数器中，是利用算法对计数器内部计数变量（例如count）加一。而在移位寄存器中，只要对寄存器移位1bit即可，具体来说，就是将内部寄存器变量（shift_reg）的最低$(n-1)$位赋值给其高$(n-1)$位，同时将输入信号din赋值给寄存器的最低位。用VHDL实现如下：

```
shift_reg := shift_reg(n-2 downto 0) & din;
```

最后一步与基本计数器相似，也就是用标准的信号赋值语句将内部变量shift_reg的值赋给输出信号。在移位寄存器中，不需要进行强制类型转换，信号类型本身就是std_logic_vector，如下列代码：

```
q <= shift_reg;
```

## 19.5 约翰逊计数器

234

约翰逊计数器是对移位寄存器的简单扩展。两者之间的唯一差异是，约翰逊计数器将最低有效位取反后反馈到寄存器的最高有效位。与拥有$2^n$个状态的传统二进制计数器不同，约翰逊计数器有$2^n$个状态。这具有一些优点，但缺点是约翰逊计数器会产生额外的寄生计数器。换句话说，当$2^n$个计数器状态在工作时，另一个状态机也同时在工作，只不过它使用的是其他未被使用的二进制状态。

这种计数器有一个潜在的问题，如果在计数过程中由于错误、噪声或者其他干扰造成约翰逊计数器进入非标准状态（即约翰逊计数器有效状态之外的状态），那么在没有复位的情况下，它将无法返回到正确的约翰逊计数器状态。正常的约翰逊计数器状态序列如下表所示：

| 计数值 | $Q(3:0)$ |
|---|---|
| 0 | 0000 |
| 1 | 1000 |
| 2 | 1100 |
| 3 | 1110 |
| 4 | 1111 |
| 5 | 0111 |
| 6 | 0011 |
| 7 | 0001 |

对shift_register函数稍加修改，即可实现一个简单的约翰逊计数器，相应的VHDL代码如下：

```
library ieee;
use ieee.std_logic_1164.all;

entity johnson_counter is
  generic (
       n : integer := 4);
  port (
       clk : in std_logic;
       rst : in std_logic;
       din : in std_logic;
       q : out std_logic_vector((n-1) downto 0)
  );
end entity;
architecture simple of Johnson_counter is
begin
  process(clk, rst)
    variable j_state : std_logic_vector((n-1) downto 0);
  begin
    if rst= '0' then
      j_state := (others => '0');
    elsif rising_edge(clk) then
      j_state := not j_state(0) & j_state(n-1 downto 1);
    end if;
    q <= j_state;
  end process;
end architecture simple;
```

235

注意，现在的拼接操作是将内部状态变量j_state的最低有效位j_state[0]取反后放在下一个状态的最高有效位，然后当前状态向下移一位。

还有一点也值得注意，这个计数器没有对错误状态进行任何检查。在实际的设计中，最好是包含对无效状态的检查功能，一旦出现无效状态就对计数器复位。最坏的情况是，在恢复到正常的约翰逊计数器状态序列前，计数器出现7个时钟周期的错误。

## 19.6 BCD计数器

BCD计数器，即二进制表示十进制计数器，也就是当计数值达到十进制数10（而不是用于4位二进制计数器的正常值15）时复位的简单计数器。这种计数器常用于十进制显示和其他人机接口硬件。BCD计数器的VHDL代码与基本二进制计数器非常相似，只不过最大计数值为10（十六进制数0xA）而不是15（十六进制数0xF）。简单的BCD计数器VHDL代码如下所示。唯一的变化是，计数器检查发现计数值大于9时会自动复位（计数范围是0~9）。

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity counter is
  generic (
```

[236]

```
        n : integer := 4);
  port (
        clk : in std_logic;
        rst : in std_logic;
        output : out std_logic_vector((n-1) downto 0)
  );
end;

architecture simple of counter is
begin
  process(clk, rst)
  variable count : unsigned((n-1) downto 0);
  begin
  if rst= '0' then
    count := (others => '0');
  elsif rising_edge(clk) then
    count := count + 1;
    if count > 9 then
          count := 0;
    else if
  end if;
  output <= std_logic_vector(count);
 end process;
end;
```

## 19.7 小结

本章讨论了一些基本的计数器，并说明了如何用VHDL执行算术功能或者逻辑功能，以便得到需要的计数序列。以这些基本计数器为基础，几乎可以开发出无数种各类计数器，这留给读者自己去研究。

读者可以做一个很有用的练习，就是修改基本二进制计数器，增加一个上计数/下计数标志，计数器根据这个标志分别递增或递减。

[237]

# 第20章　锁存器、触发器和寄存器

## 20.1　引言

存储元件具有多种类型，它们在VHDL中的表示方法也各不相同，因此准确地理解这些类型非常重要，这样，综合电路时才能得到正确的结果。因为对VHDL语言各种结构能够综合成什么电路存在理解上的误差，所以设计出来的硬件经常出现缺陷（bug）。在这一章中，我们将介绍VHDL中使用的3种主要存储元件类型，它们能够被综合到FPGA平台中，这3种类型分别是锁存器、触发器和寄存器。

## 20.2　锁存器

锁存器可以简单地定义成一种电平敏感的元件。换句话说，输出仅仅依赖于输入的值。有几种不同类型的锁存器，最常用的是D锁存器和SR锁存器。

首先考虑一个简单的D锁存器，如图20-1所示。这种类型的锁存器，其输出端*Q*仅仅在使能端En为高时才随输入端*D*变化。我们在这里所说的D锁存器，完整的定义应该为电平敏感D锁存器。本书中任何时候提到锁存器，都是指电平敏感锁存器。



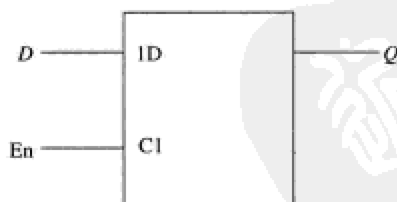



图20-1　D锁存器符号

使能信号C1和数据输入1D是联系在一起的。同时也要注意，输出*Q*完全依赖于输入*D*的电平和使能信号。换句话说，当使能信号为高时，*Q*等于*D*。这就被称为电平敏感锁存器。

以下的VHDL代码描述了这种电平敏感D锁存器：

```
library ieee;
use ieee.std_logic_1164.all;
entity latch is
  port ( d :  in std_logic;
         en : in std_logic;
         q :  out std_logic);
end entity latch;

architecture beh of latch is
begin
     process (d,en) is
```

```
    begin
        if (en = '1') then
        q <= d;
        end if;
    end process;
end architecture beh;
```

这是一个不完整if语句的例子，只有条件if (en = '1')，而没有else。信号d和en都在敏感列表中，因此这是一个组合逻辑，但是由于en条件语句不完整，结果出现了一个隐含的锁存器，也就是存储电路。

当我们开发模型时，尤其是行为级模型（这种情况下不会显式定义具体结构），这方面的信息非常重要，因为在设计中最终得到的可能是锁存器，而我们会认为创建的模型是一个纯组合电路。

另一种情况下也可能出现这样的问题，那就是不完整的case语句。例如，以下这个简单的VHDL例子：

```
case s is
  when "00" => y <= a;
  when "10" => y <= b;
  when others => null;
end case;
```

239

在这个例子中，case语句是不完整的，所以综合后生成的不是组合电路，而是锁存器电路。综合后的电路如图20-2所示。

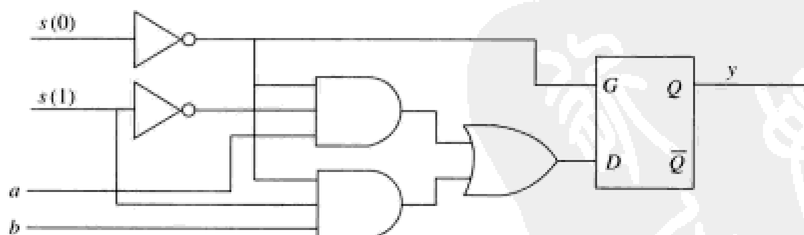



图20-2 综合后的锁存器电路

## 20.3 触发器

和电平触发的锁存器不同，触发器（flip-flop）仅仅在使能信号或时钟信号的变化边缘才会发生状态转换。这是同步设计的基础，D型触发器是一种重要的模块，如图20-3所示。输出$Q$在时钟上升沿获得输入$D$的值。图中的三角形表示时钟信号，三角形前面没有圆圈表示在时钟上升沿激活触发器，如果前面有圆圈，就表示在时钟下降沿激活触发器。

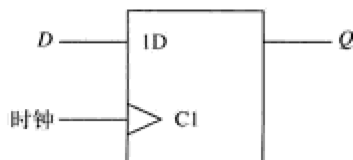



图20-3 D型触发器

对应的VHDL代码如下所示：

```
library ieee;
use ieee.std_logic_1164.all;
entity dff is
  port ( d :   std_logic;
         clk : in std_logic;
         q :   out std_logic);
end entity dff;

architecture simple of dff is
begin
process (clk)is
  begin
     if rising_edge(clk) then
       q <= d;
     end if;
end process;
end architecture simple;
```

240

需要注意，这种情况下，d并没有出现在敏感列表中，因为这是不必要的。触发器仅仅在时钟上升沿才会有所动作。描述这一功能可以用许多种不同的方法，但是它们都是等价的。在敏感列表中需要显式地定义时钟信号。另一种方法可以不使用敏感列表，但是需要在过程中增加一条wait on语句。相应的构造体如下：

```
architecture wait_clk of dff is
begin
     process is
     begin
          if rising_edge(clk) then
                q <= d;
          end if;
          wait on clk;
     end process;
end architecture simple;
```

我们也可以用一种更加复杂的方法定义上升沿功能（可能所有的仿真器和综合工具都支持）。另一种简单的方法是使用敏感列表中的时钟信号，用上升沿时，检查时钟是否为1，用下降沿时，检查是否为0。下面是上升沿D触发器的VHDL代码。注意，我们使用了隐含敏感列表（用wait on clk语句），而不是显式敏感列表，尽管两者都可以使用。

```
architecture rising_edge_clk of dff is
begin
  process is
  begin
    if (clk = '1') then
          q <= d;
    end if;
    wait on clk;
```

```
  end process;
end architecture simple;
```

我们可以将这个基本的D触发器模型扩展为带异步置位和复位功能的D触发器。所谓异步置位和复位，是指不管有没有时钟沿都会置位或复位，因此，置位和复位这两个信号需要增加到模型的敏感列表中。

241

置位和复位为低有效的触发器符号如图20-4所示。

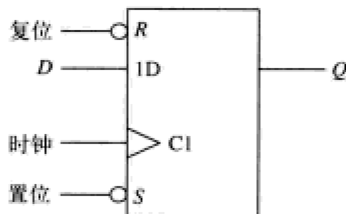



图20-4　带有异步置位和复位的D触发器

将前面给出的简单dff模型扩展为有异步置位和复位的D触发器，其VHDL代码如下：

```
library ieee;
use ieee.std_logic_1164.all;
entity dff_sr is
  port ( d    : in std_logic;
         clk  : in std_logic;
         nrst : std_logic;
         nset : in std_logic;
         q    : out std_logic);
end entity dff_sr;

architecture simple of dff_sr is
begin
process (clk, nrst, nset) is
  begin
    if (nrst = '0') then
     q <= '0';
    elsif (nset = '1') then
     q <= '1';
    elsif rising_edge(clk) then
     q <= d;
    end if;
end process;
end architecture beh;
```

对于基本的D触发器，可以使用多种方式检查时钟的上升沿。事实上我们有3个可能的输入状态控制变量（set、reset和clk），但现在还无法检查时钟是否为高（对于上升沿触发器）。但是，检查时钟为高和事件是否发生是必须的。

需要注意一点，这个模型在综合时可能会引起异常现象，因为复位信号总是在置位信号之前被检查，所以尽管从功能上来说允许同时置位和复位，但实际上复位却具有优先权。

242

最后，考虑一下0和1之间的转换。当使用不同的方法时，综合与仿真可能会有问题。例

如，使用标准逻辑包（变量为std_logic类型）的情况下，转换是严格定义的，所以在转换期间可能出现高阻或者无关状态的情况。这时，上升沿和下降沿功能就非常有用，因为它们将许多选项简化为一个单一的功能，可以很清楚地处理所有可能的转换状态。

因此，一般情况下，只要可能，最好使用上升沿和下降沿功能来保证模型之间的操作一致性和协调性。

另外也要考虑一下同步置位和复位，这样在设计中只要考虑时钟的上升沿或下降沿即可。这样做时唯一需要注意到地方是，应该在时钟边沿后立即检查置位和复位信号，以免置位和复位信号的边沿同时出现。

## 20.4　寄存器

装载和存储总线数据的一组触发器构成寄存器。寄存器和触发器的区别是，寄存器具有数据输入端、时钟，通常还有复位端，也有一个load信号，用来指示输入端数据是否装载到寄存器内部。下面的VHDL代码描述的是一个8比特寄存器：

```
library ieee;
use ieee.std_logic_1164.all;
entity register is
  generic (n : natural := 8);
  port ( d : in std_logic_vector(n-1 downto 1);
         clk : in std_logic;
         nrst : in std_logic;
         load : in std_logic;
         q : out std_logic_vector(n-1 downto 1));
end entity register;

architecture beh of register is
begin
process (clk, nrst) is
  begin
    if (nrst = '0') then
        q <= (others => '0');
    elsif (rising_edge(Clock) and (load = '1')) then
        q <= d;
    end if;
  end process;
end architecture beh;
```

243

注意，尽管有4个输入信号（clk、nrst、load和d），但只有时钟clk和复位nrst包含在过程的敏感列表内。如果load信号和数据d发生变化，那么过程将忽略它们，直到clk的上升沿到来或者nrst变为低电平。如果不使用load，寄存器在每个时钟上升沿都将输入数据装载进来，除非复位信号为低。这个寄存器比前一个稍微简单一点，VHDL代码如下：

```
library ieee;
use ieee.std_logic_1164.all;
entity reg_rst is
  port ( d, clk, nrst : in std_logic;
```

```
        q : out std_logic);
end entity reg_rst;

architecture beh of reg_rst is
begin
process (clk, nrst) is
  begin
    if (nrst = '0') then
       q <= '0';
    elsif rising_edge(clk) then
       q <= d;
    end if;
end process;
end architecture beh;
```

## 20.5 小结

本章描述了锁存器和寄存器的基本类型，并给出了相应的例子。这是同步数字系统的基本组成单元，也是RTL设计的基础。

244



# 第21章　串并转换与并串转换

## 21.1　串并转换

串并转换（Serial to Parallel Conversion，SIPO）是相当简单的事情：在时钟驱动下，将单比特的位流输入寄存器中，并依次逐位移动，直到寄存器满了为止，然后直接读取并行的输出即可。本章给出了一个串并转换的VHDL模型，寄存器的大小使用generic（n）设置，默认值为8。需要注意，这个例子中复位信号nrst是同步的，而不是以前那样的异步信号。这样，唯一触发process的信号是时钟clk的上升沿事件。当这个事件发生时，检查复位信号是否为低，不为低时寄存器由时钟驱动，为低时，寄存器被清零。

```
LIBRARY ieee;
USE ieee.Std_logic_1164.ALL;
USE ieee.Std_logic_unsigned.ALL;

ENTITY sipo IS
      GENERIC(n : Positive := 8);
      PORT(
                  clk : in std_logic;
                  nrst : in std_logic;
                  di : in std_logic;
                  q : out std_logic_vector((n-1) DOWNTO 0)
            );
END sipo;

ARCHITECTURE simple OF sipo IS
   SIGNAL int_reg : Std_logic_vector((n-1) DOWNTO 0);
   signal index : integer := 0;
BEGIN
   out_process : PROCESS
BEGIN
       WAIT UNTIL rising_edge(clk);
       if nrst = '0' then
              int_reg <= "00000000";
              index <= 0;
       else
              int_reg(index) <= di;
              if index = 7 then
                     index <= 0;
              else
                    index <= index + 1;
```

```
                end if;
        end if;
  END PROCESS;
  q <= int_reg;
END simple;
```

## 21.2 并串转换

并串转换操作包含两个阶段。第一个阶段是载入并行的数据。在以下模型中，load信号为同步信号，且为低有效。换句话说，就像串并转换模型一样，没有异步功能，时钟是敏感表内的唯一信号。如果load为高，那么寄存器内的数据被时钟逐位移出。注意，并串转换（Parallel to Serial Conversion，PISO）模型的循环不会在所有数据输出之后停止。

```
LIBRARY ieee;
USE ieee.Std_logic_1164.ALL;
USE ieee.Std_logic_unsigned.ALL;

ENTITY piso IS
      GENERIC(n : Positive := 8); -- size of register
      PORT(
              clk : IN Std_logic;
             load : IN std_logic;
              do :  OUT std_logic;
               q :  IN Std_logic_vector((n-1) DOWNTO 0));
END piso;

ARCHITECTURE simple OF piso IS
   SIGNAL int_reg : Std_logic_vector((n-1) DOWNTO 0);
   SIGNAL index : integer := 0;
BEGIN
   out_process : PROCESS
BEGIN
    WAIT UNTIL rising_edge(clk);
    if load = '0' then
           int_reg <= q;
           index <= 0;
    else
           do <= int_reg(index);
           if index = 7 then
                  index <= 0;
           else
                  index <= index + 1;
           end if;
    end if;
  END PROCESS;
END simple;
```

246

## 21.3 小结

本章简短地说明了一个有用的功能：串并转换和并串转换。在现代FPGA接口中，这是一个非常普遍的功能，大多数通信数据为串行方式，而大多数处理器则要求数据以并行方式存储和处理。

# 第22章　ALU功能

## 22.1　引言

微处理器的核心部分是ALU（Arithmetic Logic Unit，算术逻辑单元）。顾名思义，这个模块从寄存器中获得输入数据，并执行相关逻辑功能，例如非（NOT）、与（AND）、或（OR）和异或（XOR），也可能执行算术功能，例如加法或减法。本章将描述如何用VHDL实现这些底层逻辑和算术功能。

## 22.2　逻辑功能

考虑用VHDL实现一个简单的反相器，输入为1比特，反相后输出。VHDL代码如下：

```
Library ieee;
Use ieee.std_logic_1164.all;
Entity inverter is
     Port (
           A : in std_logic;
           Q : out std_logic
     );
End entity inverter;
Architecture simple of inverter is
Begin
     Q <= NOT A;
End architecture simple;
```



输入和输出定义为std_logic类型，方向分别为in和out。逻辑等式也是直接实现的。我们只要将输入和输出类型从std_logic改为std_logic_vector，就可以扩展以上代码为$n$比特模式，仅仅改变实体名称，构造体名称不变，代码如下：

```
Library ieee;
Use ieee.std_logic_1164.all;
Entity bus_inverter is
     Port (
           A : in std_logic_vector(15 downto 0);
           Q : out std_logic_vector(15 downto 0)
     );
End entity bus_inverter;
Architecture simple of bus_inverter is
Begin
     Q <= NOT A;
End architecture simple;
```

从以上代码可以看出，总线宽度为16bit，这对于具有固定结构的处理器而言较为合适，但有时候总线宽度可配置对于更一般的情况却更有用。这种情况下，我们可以继续修改实体代码，使总线宽度成为模型的一个参数：

```
Library ieee;
Use ieee.std_logic_1164.all;
Entity n_inverter is
     Generic (
          N : natural := 16
     );
     Port (
          A : in std_logic_vector((n-1)downto 0);
          Q : out std_logic_vector((n-1)downto 0)
     );
End entity n_inverter;
Architecture simple of n_inverter is
Begin
     Q <= NOT A;
End architecture simple;
```

当然，我们可以创建不同的模型来分别实现多种逻辑功能，也可以创建一个具有一组配置引脚的多功能逻辑模块，通过配置引脚可以实现不同的逻辑功能。如果我们定义了一个通用逻辑模块，具有两个$n$比特输入$A$和$B$、一个控制总线$S$和一个$n$比特输出$Q$，那么通过设置2bit的控制信号$S$，就可以根据下表选择适当的逻辑功能： 249

| $S$ | 逻辑功能 |
|---|---|
| 00 | $Q$ <= NOT $A$ |
| 01 | $Q$ <= $A$ AND $B$ |
| 10 | $Q$ <= $A$ OR $B$ |
| 11 | $Q$ <= $A$ XOR $B$ |

显然，我们还可以定义更多的逻辑功能，这需要使用更多位的选择信号$S$，不过以上这些有限的逻辑功能已经完全可以说明相关的原理了。我们对原来的实体修改如下：

```
Library ieee;
Use ieee.std_logic_1164.all;
Entity alu_logic is
    Generic (
         N : natural := 16
    );
    Port (
         A : in std_logic_vector((n-1) downto 0);
         B : in std_logic_vector((n-1) downto 0);
         S : in std_logic_vector(1 downto 0);
         Q : out std_logic_vector((n-1) downto 0)
    );
End entity alu_logic;
```

现在，根据输入选择信号*S*的值，可以选择不同的逻辑功能。可以使用第1章中介绍的case语句来定义*S*的每一个状态以及执行哪一个逻辑功能，这种语句结构非常紧凑、简洁。代码如下：

```
Architecture basic of alu_logic is
Begin
    Case S is
        When "00" => Q <= NOT A;
        When "01" => Q <= A AND B;
        When "10" => Q <= A OR B;
        When "11" => Q <= A XOR B;
    End case;
End architecture basic;
```

显然，对于控制信号*S*的每一个状态而言，这是一种定义其组合逻辑的简洁而有效的方法，但是每次赋值都必须非常小心，以避免在综合时引入不必要的锁存器。

250

## 22.3 1位加法器

ALU的核心算法是加法，使用加法器实现。我们从简单的1位加法器开始，然后再扩展到多位，直至ALU中需要的任何位数的加法器。1位加法器有两个输入端*a*和*b*，以及一个和输出端sum和一个进位输出端carry，其真值表如下：

| *a* | *b* | sum | carry |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 |

进位逻辑可以使用2输入与门（AND）实现，而求和功能可以使用2输入异或门（XOR）实现，如图22-1所示。

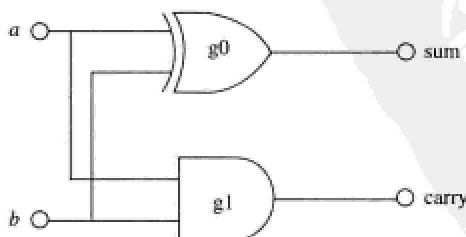



图22-1 一位加法器

这个简单的加法器具有进位输出信号（carry），却没有进位输入，为了将这个加法器扩展为多位加法器，需要实现一个进位输入功能（cin）和一个进位输出功能（cout），等效的逻辑功能如图22-2所示。

251

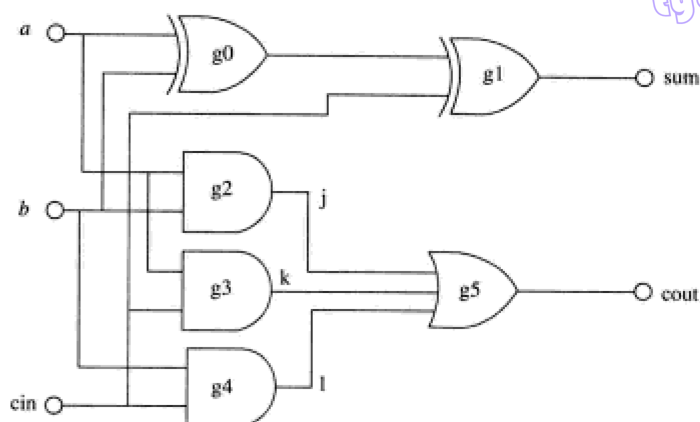



图22-2 带进位输入和进位输出功能的一位加法器

| a | b | cin | sum | Cout |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

实现时可以使用具有如下输入和输出的标准VHDL逻辑函数。首先用bit类型定义实体的输入输出端口：

```
entity full_adder is
       port (sum, co : out bit;
             a, b, ci : in bit);
end entity full_adder;
```

然后在模型的构造体中可以使用VHDL的标准内建逻辑功能，逻辑等式用来定义行为，模型中不带有任何延迟。

```
architecture dataflow of full_adder is
begin
  sum <= a xor b xor ci;
  co <= (a and b) or
        (a and ci) or
        (b and ci);
end architecture dataflow;
```

现在，这个模型就成为一个简单的构造块了，我们只要在结构上将多个构造块连接在一起，就可以创建多位的加法器。

## 22.4 n位结构化加法器

252

使用前面定义的1位全加器可以很容易地创建一个多位全加器。举一个例子，创建一个4位加法器，具有进位输入和进位输出，VHDL代码如下：

```
entity four_bit_adder is
  port (sum : out bit_vector (3 downto 0); co : out bit;
        a, b : in bit_vector (3 downto 0); ci : in bit);
end entity four_bit_adder;

architecture simple of four_bit_adder is
  signal carry : bit_vector (3 downto 1);
begin
  fa0 : entity work.full_adder
      port map (sum(0), carry(1), a(0), b(0), ci);
  fa1 : entity work.full_adder
      port map (sum(1), carry(2), a(1), b(1), carry(1));
  fa2 : entity work.full_adder
      port map (sum(2), carry(3), a(2), b(2), carry(2));
  fa3 : entity work.full_adder
      port map (sum(3), co, a(3), b(3), carry(3));
end architecture simple;
```

很明显，只要按照需要的位数对构造体中使用的元件进行复制，即可扩展为更多位的加法器。

## 22.5 n位可配置加法器

虽然结构化方法很有用，但却显得有些笨拙，很难根据需要重新配置。更可取的方法是在模型中增加一个generic语句，使位宽可以客户化定制。例如，如果定义一个实体，并增加两个std_logic_vector变量（前面使用的是bit_vector类型），那么这个实体将具有如下形式：

```
library IEEE;
use IEEE.std_logic_1164.all;

entity add_beh is
  generic(top : natural := 15);
    port (a : in std_logic_vector (top downto 0);
          b : in std_logic_vector (top downto 0);
          cin : in std_logic;
          sum : out std_logic_vector (top downto 0);
          cout : out std_logic);
end entity add_beh;
```

从这个实体中可以看出，新定义的参数top规定了输入向量（*a*和*b*）以及输出向量（sum）的大小。这样，我们就可以利用最初定义1位加法器时的逻辑等式，再增加一些行为VHDL语句，创建更多的可读性模型：

253

```
architecture behavior of add_beh is
begin
  adder : process(a,b,cin)
            variable carry : std_logic;
            variable tempsum : std_logic_vector(top
            downto 0);
          begin
            carry := cin;
            for i in 0 to top loop
                 tempsum(i) := a(i) xor b(i) xor carry;
                 carry := (a(i) and b(i))
                          or (a(i) and carry)
                          or (b(i) and carry);
          end loop;
     sum <= tempsum;
     cout <= carry;
  end process adder;
end architecture behavior;
```

这个构造体说明了如何在单个过程（process）中（敏感列表为*a*、*b*、cin）封装加法操作。当*a*、*b*或cin发生变化时，过程激活。for循环用来计算临时和（tempsum），每次计算1比特，逐渐递增直到循环结束，最后的值赋给输出sum。另外，也计算了逐级的进位，每次循环使用一次。循环结束之后的进位值构成最终的进位输出。

## 22.6 2的补码

数字逻辑设计中，减法的整数部分使用的是“2的补码”形式。这样，我们就可以使用加法器计算减法，而不用单独的减法功能。2的补码是对前面介绍的“1的补码”的一种扩展。

如果考虑一个4位无符号系统，那么数的范围是0~15（二进制为0000~1111）。但是，如果是一个有符号系统，最高有效位（MSB）则代表符号（+或−），因此数的范围变为−8~+7。在二进制逻辑中，将数字从正变为负的方法很简单，只有两个步骤：首先对原数按位取反，然后再加一。

254

举一个例子。假如数字为二进制的0011。若为有符号数，因为MSB为0，所以该数为正，低3比特直接转换为十进制数3。为了得到2的补码（即−3），首先取反，可得1100，然后加一，得到最终的2的补码形式的值1101。为验证这个数是否为真正的相反数，只要将0011和其补码1101相加即可，结果应该为0000。

用VHDL实现这一功能的代码如下：

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity twoscomplement is
  generic (
```

```
        n : integer := 8
    );
    port (
        input : in std_logic_vector((n-1) downto 0);
        output : out std_logic_vector((n-1) downto 0)
    );
end;

architecture simple of twoscomplement is
begin
    process(input)
        variable inv : unsigned((n-1) downto 0);
    begin
        inv := unsigned(NOT input);
        inv := inv + 1;
        output <= std_logic_vector(inv);
    end process;
end;
```

从VHDL代码中可以看出，首先使用逻辑函数NOT取反，然后转换为无符号数，再加一(inv + 1)，最后将结果转换回std_logic_vector类型。另外需要注意一点，通过参数*n*可以对模型进行配置，以便用于不同的数据宽度。在这个例子中，使用测试平台来检查功能的正确性，其中用了两个被测电路，一个做一次转换，另一个再做一次转换，然后检查后者的输出是否与第一个的输入相同。但是这种测试并不能保证功能的正确性，因为两次转换可能存在同样的bug，不过它对简单的快速检查却很有用，生成测试数据后，很容易就可以将输入数据和最后的输出数据进行异或，检查两者之间的差异：

255

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity twoscomplementtest is
end twoscomplementtest ;

architecture stimulus of twoscomplementtest is
    signal rst : std_logic := '0';
    signal clk : std_logic := '0';
    signal count : std_logic_vector (7 downto 0);
    signal inverse : std_logic_vector (7 downto 0);
    signal check : std_logic_vector (7 downto 0);
    component twoscomplement
        port(
            input : in std_logic_vector(7 downto 0);
            output : out std_logic_vector(7 downto 0)
        );
    end component;
```

```
    for all : twoscomplement use entity
      work.twoscomplement ;
begin
    CUT1: twoscomplement port map(input => count,
      output => inverse);
    CUT2: twoscomplement port map(input => inverse,
      output => check);

    -- clock and reset process
    clk <= not clk after 1 us;
    process
    begin
      rst <= '0','1' after 2.5 us;;
      wait;
    end process;

    -- generate data
    process(clk, rst)
      variable tempcount : unsigned(7 downto 0);
begin
  if rst = '0' then
    tempcount := (others => '0');
  elsif rising_edge(clk) then
     tempcount := tempcount + 1;
  end if;
  count <= std_logic_vector(tempcount);
  end process;
end;
```

256

## 22.7 小结

本章介绍了处理器中ALU的关键模块。无论是设计人员需要从零开始设计一个完整的ALU，还是完全为了理解现有架构的行为，在分析ALU和处理器的行为时，这些功能模块都是非常有用的。

257

# 第23章　译码器与多路复用器

## 23.1　译码器

译码器是一种简单的组合逻辑电路，可以将某一种形式的数字表示转换为另一种。通常，译码器以较小的数字表示为输入，将它转换为更大的一种（编码与此相反）。典型的例子是，将一个$n$比特的输入译码为$2^n$个独立的逻辑信号。例如，3–8译码器输入3个逻辑信号，将它们转换为8（$2^3$）个输出信号中的一个，即8个输出信号中只有一个表示当前被选择的输入。这样的译码器，其符号如图23-1所示，功能如下表。

| s2 | s1 | s0 | q7 | q6 | q5 | q4 | q3 | q2 | q1 | q0 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |



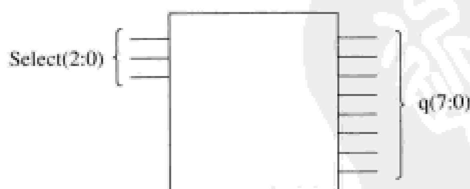



图23-1　3-8译码器

这个译码器的VHDL代码中使用了一种简单的VHDL结构，类似于if – else – end if语句，只不过使用的语法是when – else。当某个条件满足时，对一个信号赋值，则可以使用以下代码完成单个赋值：

```
output <= value when condition;
```

用else语句将上面的语句扩展，可以覆盖更多不同的条件，结果如下：

```
output <=  value1 when condition1 else
           value2 when condition2 else
               ...
           valuen when condition;
```

最后，如果有一个“覆盖所有（catch all）”的条件，即类似于VHDL中条件语句if–elsif–else–endif中最后一个else，那么增加的最后一条语句如下：

```
output <=  value1 when condition1 else
           value2 when condition2 else
               ...
           valuen when conditionn else
           valuedefault;
```

用这种方法，就可以用如下的VHDL代码简单地实现3-8译码器：

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity decoder38 is
  port (
        s : in std_logic_vector (2 downto 0);
        q : out std_logic_vector(7 downto 0)
  );
end;

architecture simple of decoder38 is
begin
        q <= "00000001" when s = "000" else
        "00000010" when s = "001" else
        "00000100" when s = "010" else
        "00001000" when s = "011" else
        "00010000" when s = "100" else
        "00100000" when s = "101" else
        "01000000" when s = "110" else
        "10000000" when s = "111" else
        "XXXXXXXX";
end;
```

259

这个译码器的测试平台可能是一个简单的数值的查找表，不过，实际上我们可以结合计数器例子中的时钟和复位测试平台以及测试平台中的简单计数器为译码器产生一些输入信号，具体如下：

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity Decoder38Test is
end Decoder38Test;

architecture stimulus of Decoder38Test is
      signal rst : std_logic := '0';
      signal clk : std_logic := '0';
      signal s : std_logic_vector(2 downto 0);
      signal q : std_logic_vector(7 downto 0);
```

```
        component decoder38
              port(
                    s : in std_logic_vector(2 downto 0);
                    q : out std_logic_vector(7 downto 0)
              );
        end component;
        for all : decoder38 use entity work.decoder38;
  begin
        CUT : decoder38 port map(s => s, q => q);
        clk <= not clk after 1 us;
        process
        begin
         rst <= '0','1' after 2.5 us;
         wait;
        end process;
  process(clk,rst)
        variable count : unsigned(2 downto 0);
  begin
        if rst = '0' then
           count := (others => '0');
        elsif rising_edge(clk) then
           count := count + 1;
        end if;
        s <= std_logic_vector(count);
  end process;

  end;
```

## 23.2 多路复用器

260

多路复用器是对简单译码器的一种扩展，是对一系列控制信号进行译码，由此产生选择信号，选择输入信号中的一个作为输出。在译码器中，$n$比特可以对$2^n$个信号译码，与此类似，在多路复用器中，$n$比特选择线可以对$2^n$个信号进行多路复用。例如，考虑一个最简单的多路复用器，两个输入信号（$A$和$B$）、一个输出信号（$Q$）和一个选择信号线（$S$）。这种多路复用器（MUX）的IEEE符号如图23-2所示。

译码器中使用when–else结构，类似地，在多路复用器中，也可以使用这种结构，相应的VHDL代码如下：

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity mux21 is
  port(
        s : in std_logic;
```

```
        a : in std_logic;
        b : in std_logic;
        q : out std_logic
  );
end;

architecture simple of mux21 is
begin
        q <= a when s = '0' else
        b when s = '1' else
        'X';
end;
```

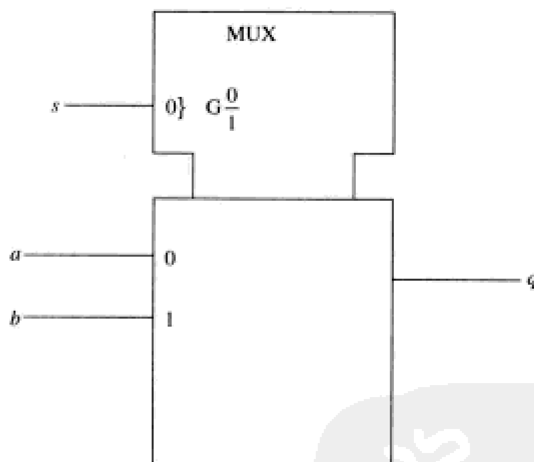



图23-2 两输入单选择线多路复用器

这是一个非常有用的模型，广泛应用于测试结构中，也就是在功能信号和测试信号之间选择一个作为触发器的输入。这一模型很容易扩展以提供多个输入信号。例如，考虑一个四输入、两根选择线(inputs=2select)和一个输出的多路复用器。相应的VHDL代码如下： 261

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity mux21 is
  port (
        s : in std_logic_vector (1 downto 0);
        a : in std_logic;
        b : in std_logic;
        c : in std_logic;
        d : in std_logic;
        q : out std_logic
  );
end;
```

```
architecture simple of mux21 is
begin
        q <= a when s = "00" else
        b when s = "01" else
        c when s = "10" else
        d when s = "11" else
        'X';
end;
```

## 23.3 小结

本章简短地描述了用VHDL构造译码器和多路复用器的基本原理。这是一个非常有用的功能，因为在FPGA中，可以用它来管理大量的数据和控制信号。

262



# 第24章　VHDL中的有限状态机

## 24.1　引言

有限状态机（Finite State Machines，FSM）是大多数数字设计的核心。有限状态机的基本概念是：存储一系列不同的状态，根据输入和当前状态在这些状态之间进行转换。有限状态机有两种类型，分别是Moore型（状态机的输出完全由状态变量决定）和Mealy型（状态机输出既与当前状态变量有关，又与输入有关）。有限状态机的一般结构如图24-1所示。

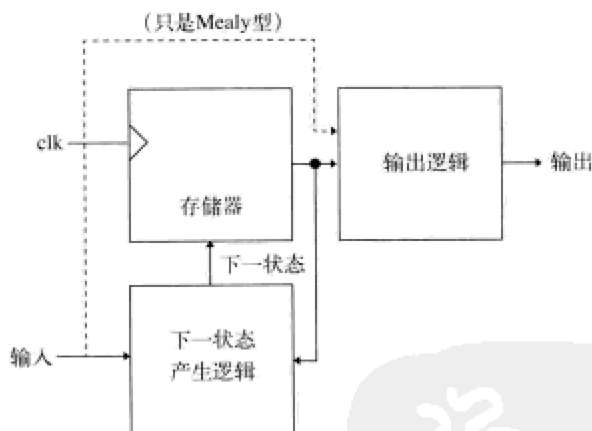



图24-1　硬件状态机结构

## 24.2　状态转移图

从设计的观点看，描述状态机的方法是使用状态转移图（泡泡图），它可以表示状态、输出和转移条件。图24-2是一个简单的状态转移图。



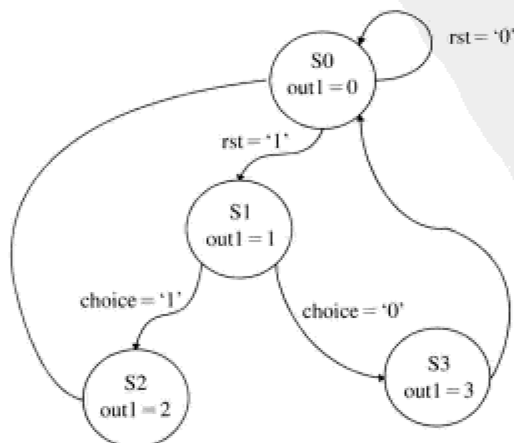



图24-2　状态转移图

状态转移图说明如下。有4个泡泡（状态）。其间的转换由rst和choice两个信号控制，这两个信号都可以是bit或者std_logic类型（或其他类似的逻辑类型）。还有一个隐含的时钟信号，我们称其为clk，输出信号为out1。

## 24.3 用VHDL实现有限状态机

可以用process中的一条case语句实现这个状态转移图，VHDL代码如下：

```
library ieee;
use ieee.std_logic_1164.all;

entity fsm is
  port(
       clk, rst, choice : in std_logic;
       count : out std_logic
  );
end entity fsm;
architecture simple of fsm1 is
  type state_type is ( s0, s1, s2, s3 );
  signal current,next_state : state_type;
begin
  process ( clk )
  begin
       if (clk = '1' )) then
           current <= next_state;
       end if;
  end process;
process (current)
begin
      case current is
           when s0 =>
                 out <= 0;
                 if (rst = '1') then
                       next <= s1;
                 else
                       next <= s0;
                 end if;
           when s1 =>
                 out <= 1;
                 if (choice = '1' ) then
                      next <= s3;
                 else
                      next <= s2;
                 end if;
           when s2 =>
                 out <= 2;
```

264

```
                    next <= s0;
                when s3 =>
                    out <= 3;
                    next <= s0;
          end case;
      end process;
   end;
```

## 24.4 小结

在数字电路中，有限状态机是一种设计控制算法的基本技术。本章仅仅介绍了一些有限状态机的关键概念，如果读者不熟悉数字电路设计的基本概念，强烈建议你找一本有关数字设计技术的图书，补充一下本书所描述的实际实现方法的知识。 265

# 第25章　VHDL中的定点算法

## 25.1　引言

在VHDL中，我们可以完全访问各种类型，如bit类型和布尔类型（包含两个状态“1”和“0”，或者称为“真”和“假”）、整数类型（包括正整数和自然数）以及实数类型（浮点数）。但是，我们在VHDL中使用的许多类型和语法却不能综合成实际的硬件，这实在是个巨大的遗憾。

尽管最近的研究和标准化工作取得了一定的进展，但支持用于FPGA实现的定点和浮点算法的包和库仍然极其有限。对于绝大多数FPGA应用来说，尤其是一些DSP类型的应用，一般情况下定点算法在多数情况下都足够用了。

那么，什么是定点算法呢？在FPGA设计中又该如何去使用它呢？在整数算法中，无论是无符号数，有符号数还是std_logic类型，其基础都是整数的位表示法，其中没有十进制的小数点。例如，为了表示数字23，要使用8bit，为每个二进制单元设置1bit来构成整数23。如图25-1所示。



| 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 |

16+4+2+1 = 23

图25-1　基本二进制表示

如果要表示一个负数，可以使用有符号数表示方法，即最高有效位（MSB）是符号位，如图25-2所示。实际上，正如前面ALU功能一章中讨论的那样，按位取反后，在最低有效位直接加一后就可以得到2的补码形式。

| −128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 |

−128+64+32+8+1 = −23

图25-2　负数的二进制表示

使用这种处理数据的方法，就可以扩展到“定点”表示法，只要规定小数点在什么位置即可。例如，同样使用8bit表示数字，我们可以规定小数点前面有5bit，后面有3bit。当然，这对数字的小数部分有一定限制。具体实现方法是：对小数点右侧为1的位按位置取倒数，即小数点右侧第一位对应二分之一，右侧第二位对应四分之一，依此类推。下面举例说明，

仍以图25-2中的数字为例，在新的定点数表示方法中，这个数不再是−23，而是−2.875，如图25-3所示。

| −16 | 8 | 4 | 2 | 1 | 1/2 | 1/4 | 1/8 |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 |

−16+8+4+1+0.125 = −2.875

图25-3 定点数表示

这种表示方法的好处是，前面所开发的基于整数运算的ALU位操作函数几乎不用做任何改动就可以使用这种新的定点数表示法。唯一的变化是，需要将定点数类型转换为std_logic_vector类型，而且还要考虑如何处理溢出条件。

例如，如果两个数相加，结果太大，那么定点数算法中如何处理这种情况呢？只是简单地给出一个溢出标志，然后输出结果吗？还是设置最大值并输出呢？ 267

同样，对于那些非常小的数字来说，可能会出现丢失精度的问题，那么我们只是简单地向上取整或者给出一个标志表示精度丢失了吗？这些都是设计人员需要根据不同的应用来回答的问题，但是对于本章剩余的部分，我们将使用一种简单的方法来说明基本功能是如何操作的，具体的处理细节将留给读者考虑，除非这里专门指定或讨论。

## 25.2 基本定点类型

要定义一个客户化的定点类型库，第一个任务是为数字定义一种新的类型。在标准VHDL中，与数字最接近的可综合类型是无符号数和有符号数。这两种类型是按照一定量的位定义的。大多数情况下，我们感兴趣的是直接与std_logic系统连接，所以可以以std_logic类型数组为基础定义一种新的类型。本章后续部分将只讨论有符号算法，因为从应用的角度看，有符号算法最有可能用于DSP中。

我们定义的基本类型称为fixsign，定义为无固定长度的std_logic类型数组：

```
Type fixsign is array ( integer range <> ) of std_logic;
```

以此为基础，就可以定义出一些特定的定点类型数据。例如，使用以下声明可以定义一种小数点前8位、小数点后3位的数据类型：

```
Subtype fp8_3 is fixsign ( 8 downto -3);
```

用这些新类型，可以声明一些信号，并在定点类型VHDL模型中使用：

```
Signal a1 : fp8_3;
A1 <= X"0CA";
```

这样做虽然很有用，但是也有一些局限性，因为这种类型需要很容易而快速地从一种类型转换为另一种类型。管理这一过程最简单的方法是创建一个新的包，其中不仅包含类型声明，而且还包含与该类型相关的一些函数。因此，定义如下的新包，名称为fp_pkg，最少包含下面的类型声明： 268

```
package fp_pkg is
  type fixsign is array (integer range <>) of std_logic;
```

```
  subtype fp8_3 is fixsign ( 8 downto -3);
end package;

package body fp_pkg is
end package body;
```

现在，只要将这个包编译到当前工作库中，就可以在VHDL模型中使用这个包了，按照需要调用这个包即可：

```
Use work.fp_pkg.all;
```

这样就可以访问所有的定点函数和类型了。

在这个库中，定义了两种函数类型。第一种类型是将物理类型（例如std_logic_vector）转换为新的类型或者相反。这些操作非常重要，因为它们会经过综合，最终成为硬件。第二种类型纯粹是用于调试（debug）目的以及在屏幕上显示数据。例如，将定点数据转换为实数，然后用VHDL中的real'image函数在屏幕上显示。所以本章介绍一组有用的函数。再说明一次，这些函数只是一些范例，我们鼓励读者为自己的特定应用开发专门的函数。

## 25.3 定点函数

### 25.3.1 定点数向std_logic_vector的转换

最重要的函数是定点数与std_logic_vector类型变量之间的转换函数。如果两者之间能够转换，那么就可以直接在适合用定点数的地方使用标准的逻辑功能块，而不用每次重新设计全新的模块了。

最简单的函数是从定点数映射为std_logic_vector类型变量，操作时只要从定点数的最低有效位（LSB）开始，依次将输出std_logic_vector变量的每一位设置为正确的值即可。相应的VHDL代码如下：

269

```
  function fp2std_logic_vector (d : fixsign;top:integer;
     low : integer)
        return std_logic_vector is
        variable outval : std_logic_vector (top-low
          downto 0 ) := (others => '0');
  begin
        for i in 0 to top-low loop
              outval(i) := d(i + low);
        end loop;
  return outval;
end;
```

仔细观察这个函数就可以发现，函数的参数有一个定点数和两个整数，其中两个整数分别表示小数点前的位数和小数点后的位数。例如，对于8.3的表示方法，函数调用为如下形式：

```
Q <= fp2std_logic_vector (d,8,-3);
```

需要注意一点，负数表示小数点后的位数。如果想让两个数字都为正数，只要进行简单

的修改即可。使用负数的一个原因是它满足基本类型定义，因此检查起来更加容易。

同样，还可以用类似的函数从反方向将std_logic_vector类型变量转换为定点数，如下列代码所示：

```
function std_logic_vector2fp
  (d:std_logic_vector; top : integer; low : integer)
return fixsign is
      variable outval : fixsign (top downto low )
        := (others => '0');
begin
      for i in 0 to top-low loop
            outval(i + low) := d(i);
      end loop;
      return outval;
end;
```

270

函数调用如下：

```
Q <= std_logic_vector(d,8,-3);
```

使用这些函数，std_logic_vector类型和定点数算术域之间的转换就可以直接进行了。而且，这些函数还是可综合的，因为它们只是简单地进行了位映射，而没有执行任何其他复杂的功能。

### 25.3.2 定点数向实数的转换

从定点数向实数的转换是一个极其有用的函数。很明显，这不能用于综合，但是对于测试平台的编写、检查和报告等都非常有用。因此，我们只要定义一个函数fp2real即可，它将定点数转换为实数用于显示。只要我们有数据，那么real'image函数就可以显示其值。转换函数的VHDL代码如下：

```
function fp2real (d : fixsign; top : integer; low : integer)
return real is
        variable outreal : real := 0.0;
        variable mult : real := 1.0;
        variable max : real := 1.0;
        variable debug : boolean := false;
begin
        for i in 0 to top-1 loop
              if d(i) = '1' then
                    outreal := outreal + mult;
                    if debug then
                      report " fp2real : " &
                        integer'image(i);
                    end if;
              end if;
              mult := mult * 2.0;
        end loop;
```

```
if debug then
   REPORT " fp2real middle : " & real'image(outreal);
end if;
max := mult;

mult := 0.5;

for i in -1 downto low loop
      if d(i) = '1' then
             outreal := outreal + mult;
             if debug then
               report " fp2real : " & integer'image(i);
             end if;
      end if;
      mult := mult * 0.5;
end loop;
if debug then
  REPORT " fp2real : " & real 'image(outreal);
end if;

if d(top) = '1' then
       outreal := outreal - max;
end if;
if debug then
  REPORT " fp2real FINAL VALUE : " &
      real'image(outreal);
end if;

return outreal;
end;
```

271

这个函数是一个简单的转换器，依次对小数点前后的位进行处理。也应该注意，内部的布尔型变量debug可以对每一个单独的位进行检查。在观察数据通过边界的传递情况时，这一功能非常有用。这个变量的默认值为false，即关闭调试功能。

如果需要报告一个定点数的值，可以使用这个函数，VHDL代码如下：

```
D : fp8_3;
Dr : real;
Dr <= fp2real(fp8_3,8,-3);
Report "The value is : " & real'image(Dr);
```

## 25.4 测试定点数函数

正如前文所述，我们可以使用这些函数将标准的std_logic类型ALU功能集成到模型中。这里所列的测试例子中，使用ALU功能一章中定义的标准*n*位加法器将两个定点数相加。它是怎样工作的呢？我们所要做的就是将两个输入定点数转换成std_logic_vector类型数据，并

输入给加法器模块，然后再将输出转换为定点数。为了在屏幕上观察结果，可以将输入和输出转换为实数。测试代码如下：



```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

use work.fp_pkg.all;
entity simple1 is
end entity simple1;

architecture tb of simple1 is
      signal clk : std_logic := '0';
      signal cin : std_logic := '0';
      signal cout : std_logic;
      signal testa : fp8_3 := "000000000000";
      signal testa1 : fixsign (8 downto -3);
      signal testa2 : fixsign (8 downto -3);
      signal testb1 : fixsign (8 downto -3);
      signal testsum : fixsign (8 downto -3);
      signal as : signed (11 downto 0):= X"000";
      signal a1std : std_logic_vector (11 downto 0):=
        X "800";
      signal b1std : std_logic_vector (11 downto 0):=
        X "800";
      signal sum : std_logic_vector (11 downto 0);
      signal a1out : real;
      signal b1out : real;
      signal a2out : real;
      signal sumout : real;
      signal a1 : integer := 0;
      signal bs : signed (11 downto 0)  := X"8f0";

      component add_beh
      generic (
              top : integer := 7
      );
      port (
              signal a : in std_logic_vector(top
                downto 0);
              signal b : in std_logic_vector(top
                downto 0);
              signal cin : in std_logic;
              signal cout : out std_logic;
              signal sum : out std_logic_vector(top
                downto 0)
```

```
        );
        end component;
        for all : add_beh use entity work.add_beh;
    begin
        clk <= not clk after 1 us;

        DUT : add_beh generic map ( 11 ) port map ( a1std,
          b1std, cin, cout, sum);

        p1 : process (clk)
        begin
             as <= as + 1;
             testa1 <= signed2fp(as,8,-3);
             testb1 <= signed2fp(bs,8,-3);
             a1out <= fp2real(testa1,8,-3);
             b1out <= fp2real(testb1,8,-3);
             a1std <= fp2std_logic_vector(testa1,8,-3);

             b1std <= fp2std_logic_vector(testb1,8,-3);
             testa2 <= std_logic_vector2fp(a1std,8,-3);
             testsum <= std_logic_vector2fp(sum,8,-3);
             a2out <= fp2real(testa2,8,-3);
             sumout <= fp2real(testsum,8,-3);
             report "a1out : " & real 'image(a1out);
             report "a2out : " & real 'image(b1out);
             report "sumout : " & real 'image(sumout);

        end process p1;
    end;
```

273

这个测试模型中有一点很重要，那就是信号和时钟clk的使用。通过对模型的同步化处理，我们可以保证模型具有正确的、可预测的行为，但是在每一个时钟周期都有内在的延迟。在信号sumout上观察到的最终结果（用于显示的实数输出）将比模型的输入数据晚两个时钟周期。

这个例子中，我们使用了有符号数作为原始的输入as，因为有符号数可以很容易递增，另外将输入bs设置为常数。这些输入被转换为实数（a1out和b1out），以方便在屏幕上显示结果。

## 25.5 小结

本章介绍了VHDL中的定点数算法概念，也为读者提供了一个基本的函数和类型包。但必须强调的是，这个包纯粹是为了设计范例的说明，建议读者要么使用商业库以得到最优化的性能，要么开发自己的库。

274

# 第26章　二进制乘法

## 26.1　引言

在有关信号处理的硬件设计中，有一个非常关键的功能就是乘法。为了实现这一功能，有必要从基本原理开始介绍一下二进制乘法的计算方法，从而进一步理解具体的实现方法。本章将对这些方法进行描述并用VHDL代码举例说明。

## 26.2　基本二进制乘法

实现二进制乘法最简单的方法是将长乘法（long multiplication）应用于二进制数。首先，我们举一个十进制数长乘法的例子，回忆一下基本概念，计算23与17的乘积：

```
  23
 ×17
 ---
 161
 230
 ---
 391
```

二进制数乘法也可以用完全相同的方法实现，只要用二进制数代替十进制数、二进制算法代替十进制算法即可。假设要计算两个无符号二进制数0110（十进制数6）和0100（十进制数4）的乘积。计算过程中，检查乘数（例如4）的每一位。如果该位为0，不加到结果；如果该位为1，将移位后的被乘数（例如6）加到结果中。



```
    0110   (6)
   ×0100   (4)
   -----
    0000
   0000
  0110
 0000
 -------
  011000   (24)
```

实际中实现的方法是计算“部分积”，然后将每一阶段移位后的被乘数相加，直到乘法完成。

这种方法对于无符号二进制数能够正常工作，但是不能用于2的补码。若使用2的补码，应用与此类似的方法需要在每一阶段移位后的被乘数左侧加上符号位，最后一步取消被乘数并将最终移位后的值加到部分积中。一种更简单更适合于硬件实现的方法是，检查数字是不是负数，如果是负数则转换为正数，然后进行无符号乘法，最后根据输入参数中负数的个数决定输出是否转换为2的补码形式。检查是否是负数的方法非常简单，只要对两个输入有符号数字的最高有效位（MSB）进行异或（XOR），即可知道输出是否需要转换为2的补码形式。具体过程如图26-1所示。

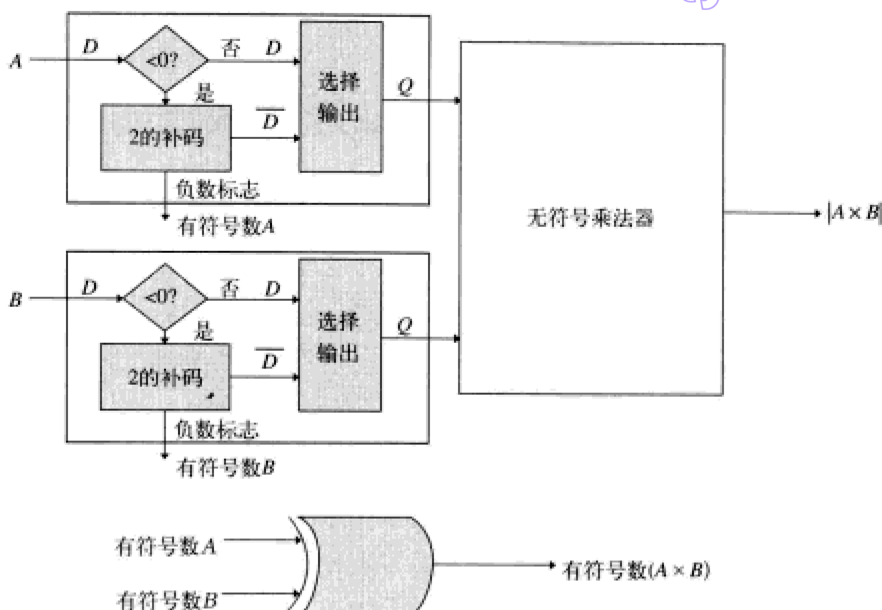



图26-1　基本有符号乘法

## 26.3　VHDL无符号乘法器

我们先从一个简单的无符号乘法器开始，因为这很容易用VHDL实现。重要的是想清楚这个乘法器需要多少位输入和多少位输出。如果输入和输出使用相同的位数，那么还要考虑溢出的可能，以及如何处理溢出的情况。在这个基本的模型中，我们将输出位数定义为两个输入字长的和，并在外部用乘法器处理溢出的情况。

276

我们可以使用前面介绍的基本累加器和二进制加法器来实现一个基本的乘法器，相应的VHDL代码如下：

```
library ieee;
use IEEE.std_logic_1164.all;

entity mult_beh is
  generic(top : natural := 15);
    port (
        clk : in std_logic;
        nrst : in std_logic;
           a : in std_logic_vector (top downto 0);
           b : in std_logic_vector (top downto 0);
           product : out std_logic_vector (2*top+1
             downto 0)
    );
end entity mult_beh;

architecture behavior of mult_beh is
```

```
        component add_beh
        generic (
              top : integer := 7
        );
        port (
              signal a : in std_logic_vector(top downto 0);
              signal b : in std_logic_vector(top downto 0);
              signal cin : in std_logic;
              signal cout : out std_logic;
              signal sum : out std_logic_vector
              (top downto 0)
        );
        end component;
        for all : add_beh use entity work.add_beh;

        signal cin : std_logic := '0';
        signal cout : std_logic := '0';
        signal acc : std_logic_vector(2*top+1 downto 0);
        signal sum : std_logic_vector(2*top+1 downto 0);
        signal mand : std_logic_vector(2*top+1 downto 0);
        signal index : integer := 0;
        signal finished : std_logic := '0';
begin
        DUT : add_beh generic map (2*top+1) port map
          (acc,mand,cin,cout,sum);

    p1 : process (clk, nrst)
      variable mandvar : std_logic_vector(2*top+1 downto 0);
    begin
            if (nrst = '0')then
                 acc <= (others => '0');
                 index <= 0;
                 finished <= '0';
            else
                 if rising_edge(clk)then
                      if index <= top then
                            index <= index [Plus] 1;
                            mandvar := (others => '0');
                            if b(index) = '1'then
                              for i in 0 to top loop
                                mandvar(i+index):= a(i);
                              end loop;
                            end if;
                      end if;
                      mand <= mandvar;
                      acc <= sum;
                 end if;
```

```
                if falling_edge(clk) then
                     if index > top-1 then
                           finished <= '1';
                     end if;
                end if;
           end if;
   end process p1;
   p2 : process (finished)
   begin
           if rising_edge(finished) then
                product <= sum;
           end if;
   end process p2;
end architecture behavior;
```

278

这个模型可能比实际需要的更加复杂，但是有利于读者学习。

首先，与追求效率而导致可读性不强的移位模型不同，上述模型中过程（process）p1对移位操作与加法操作进行了详细的安排，所以乘法的每一阶段才能顺利进行。另外还要注意信号finished的使用，该信号用来表示计算完成。当设计一个控制器来表示计算完成时，就用这个信号。

## 26.4 乘法函数的综合

完成设计之后，使用标准的综合软件对这个模型进行综合，目标器件为Xilinx公司的Virtex II Pro系列FPGA，选择一个大小合适的即可，综合结果如下：

```
Number of ports :                             66
Number of nets :                            1704
Number of instances :                       1639
Number of references to this view :            0
Total accumulated area :
Number of BUFGP :                              1
Number of Dffs or Latches :                  164
Number of Function Generators :             1181
Number of IBUF :                              33
Number of MUX CARRYs :                        31
Number of MUXF5 :                            221
Number of MUXF6 :                              2
Number of OBUF :                              32
Number of accumulated instances :           1701
Number of global buffers used:                 1
************************************************
Device Utilization for 2VP2fg256
************************************************
Resource              Used    Avail     Utilization
------------------------------------------------
```

```
IOs                       65       140       46.43%
Global Buffers             1        16        6.25%
Function Generators     1181      2816       41.94%
CLB Slices               591      1408       41.97%
Dffs or Latches          164      3236        5.07%
Block RAMs                 0        12        0.00%
Block Multipliers          0        12        0.00%
-------------------------------------------------
Clock                    : Frequency
--------------------------------
clk                      : 30.0 MHz
finished                 : 30.0 MHz
```

279

从这个报告中可以很清楚地看到，为了在标准器件中实现这个乘法器，使用了非常多的资源。这种情况下，主要是对面积进行优化，而不是速度，尽管这个设计只用了整个FPGA资源的近一半，所以，算法功能在FPGA上的实现并不总是很容易的，某些情况下在面积上已经无法实现，其中最耗资源的就是乘法器。

因此，管理设计时必须十分小心，尽量利用流水线技术，尽可能高效地使用可用的资源。不利的地方是设计变得更加复杂，一般都需要一个控制器，但最终得到的性能却比等效的DSP函数还要强。

## 26.5 "简单的"乘法

在前面的例子中我们已经看到，实现乘法操作的方法中使用了"乘法基本原理"介绍的方法，这一方法非常消耗资源和时间（大约需要*n*次移位才能完成乘法运算，这将是一个非常慢的器件）。

不过还有另外一种方法，许多现代的FPGA中都包含乘法器模块，可以直接在设计内使用。这些乘法器都是固化在FPGA内并经过优化处理的，可以直接在VHDL中使用并实现某些专门的乘法功能。

所以我们可以将std_logic_vector类型信号转换为有符号信号，然后直接使用如下的VHDL代码得到两个数的乘积（记住，*a*和*b*是两个输入，其类型都是std_logic_vector，乘积是输出，类型也是std_logic_vector）。

```
Product <= std_logic_vector( signed(a) * signed(b) );
```

很明显，这条语句比前面的模型简单并且有效得多，不过也要记得声明IEEE数字标准类型库（numeric standard library）：

```
Use ieee.numeric_std.all;
```

这样声明之后就可以使用有符号类型了。用这种方法得到的最终模型更加紧凑简洁，如下所示：

280

```
library ieee;
use IEEE.std_logic_1164.all;
use ieee.numeric_std.all;
entity mult_sign is
  generic(top : natural := 15);
```

```
    port (
        clk : in std_logic;
        nrst : in std_logic;
        a : in std_logic_vector (top downto 0);
          b : in std_logic_vector (top downto 0);
          product : out std_logic_vector
            (2*top+1 downto 0)
    );
end entity mult_sign;

architecture behavior of mult_sign is
begin
      p1 : process (a,b)
      begin
           product <= std_logic_vector(signed(a)*
             signed(b));
      end process p1;
end architecture behavior;
```

最后得到的综合输出也更加简洁。当然，所用的IO模块（IOB）数量是一样的，但FPGA内部资源的使用率则显著减少：

```
Number of ports :                              66
Number of nets :                              128
Number of instances :                          65
Number of references to this view :             0
Total accumulated area :
  Number of Block Multipliers :                 1
  Number of gates :                             0
  Number of accumulated instances :            65
Number of global buffers used:                  0
************************************************
Device Utilization for 2VP2fg256
************************************************
Resource                 Used    Avail    Utilization
------------------------------------------------
IOs                      66      140      47.14%
Global Buffers           0        16       0.00%
Function Generators      0      2816       0.00%
CLB Slices               0      1408       0.00%
Dffs or Latches          0      3236       0.00%
Block RAMs               0        12       0.00%
Block Multipliers        1        12       8.33%
```

从上表可以明显看出，这个器件中有12个乘法器可用，我们在设计中只使用了一个，所以剩余的乘法器的利用率为0。这就使得这些器件可以非常高效地实现某些低阶滤波器。

281

## 26.6 小结

本章介绍了一些在FPGA中使用VHDL实现乘法器的技术，重点强调了使用“基本原理”的方法实现乘法器与直接利用FPGA内的固化乘法器之间的区别，分别在面积和模型的复杂度方面进行了说明。

当然，还有些其他的乘法器实现结构，包括Booth乘法器等，这些乘法器广泛用在硬件设计中。鼓励读者去研究一下类似乘法器这样的硬件在实现上的不同选项之间的差异，研究如何最好地实现自己的应用。

# 第27章 参考书目

## 27.1 引言

正常情况下，一本书都会有一个参考书目，其中简单列出了参考书的书名，但是在这本书中，我决定不但要列出参考书的书名和详细的说明，而且还要给出我对参考书内容的看法，以帮助读者决定哪本书更适合他们。当然，我自己的观点是有局限性的，其他人可能并不同意我对书的简短概括，不过，它还是有希望帮助读者理解我在每一本书中哪些地方找到的参考资料。

## 27.2 VHDL参考书

*Digital Systems Design*

Mark Zwolinski著，Pearson教育出版集团出版。该书是一本极好的介绍用VHDL进行设计的入门教材。世界各地有许多大学将这本书作为大学阶段VHDL教学的教材，书中有许多很基础的例子适合学生学习。我也将这本书推荐给刚刚开始学习VHDL的FPGA工程师们。

*Designers Guide to VHDL*

Peter Ashenden著。从各个方面看，该书都可能是最全面的VHDL书籍。书中非常严格地覆盖了语法和语言知识，并配有大量例子，确实是一本不可多得的案头参考书。对于已经比较熟悉VHDL的中级读者来说，我推荐这本书。

283

*VHDL: Analysis and Modeling of Digital Systems*

Zainalabedin Navabi著，McGraw-Hill出版公司出版。该书不仅详细地说明了如何用VHDL建立数字系统模型，而且还涉及其他书通常跳过的许多时序分析方面的内容。这不是一本入门级的读物，而是专门针对那些已经对时序有深入了解的读者。

*VHDL for Logic Synthesis*

Andrew Rushton著，Wiley出版公司出版。对那些需要了解VHDL如何用于实际综合的读者来说，这是一本有用的参考书。这本书讨论了VHDL中哪些内容不能用于综合，并且解释了一些有用却有些神秘的VHDL函数是如何操作的。

## 27.3 FPGA参考书

### FPGA设计指南：器件、工具和流程

Clive "Max" Maxfield著，原版由Elsevier公司出版，中文版由人民邮电出版社图灵公司

出版，书号为978-7-115-16862-7。这是一本非常好的FPGA入门教材，介绍了一些用FPGA作为设计平台的主要概念，但是并没有陷入VHDL或者Verilog的底层细节中，而是在高层设计和底层设计之间达到了一定的平衡。这对于那些需要了解FPGA如何工作的学生来说非常有用，同时对需要了解如何在实际设计中使用FPGA的工程师们也很有帮助。

## 27.4 普通数字设计参考书

*Digital Design*

M. Morris Mano著，Prentice Hall公司出版。数字设计和计算机设计方面的优秀参考书。对于嵌入式处理器设计来说，非常有用的一点是书中讨论了高级语言、汇编语言和机器码之间的差异，另外书中还开发出了一套设计方法。对于任何开始从事处理器设计的人来说，这是一本非常有用的参考书。Mano还写过一本相关的书，名为*Computer System Architecture*，书中在这方面的描述更加详细，也同样有用。

284~285

# 索　引

索引中的页码为英文原书页码，与书中页边标注的页码一致。

## A



## B



## C



## D



## E



## F



## L

## M



## P



## R



## S

Design Recipes for FPGAs

# FPGA设计实战

FPGA技术成本低、使用灵活，已经成为应用广泛的电路设计解决方案，应用范围遍及消费电子、汽车电子、通信和工业控制等领域。

本书是一部实例丰富的实战宝典，涵盖了电路设计的完整流程，从FPGA基础知识入手，不但介绍了"FPGA做什么"，更详细解读了"用FPGA怎么做"。作者首先简洁而全面地概述了FPGA和标准设计流程的基础知识，然后详细阐述了两个典型的综合实例——高速视频监视系统和嵌入式处理器，并且给出了具体的代码框架，为设计者点明实际应用中的关键点和设计精髓。接下来，本书针对电路设计中几个最常见的关键任务，如串行通信、数字滤波器、安全系统、存储器建模等，给出了可以直接应用于实际项目的代码示例。书的最后介绍了综合和行为建模等设计后期的优化。

对于电路设计工程师和高校电气工程专业的师生来说，阅读本书是快速入门并成为设计高手的一条捷径。

**Peter Wilson** 博士，著名电子设计技术专家，IEEE 1076.1.1标准（VHDL-AMS的多能量域支持用组件）副主席，IEEE行为建模与仿真工作组技术项目组长。现任教于英国南安普敦大学，同时担任美国阿肯色大学客座教授。其主要研究领域为建模与仿真，先后发表过百余篇颇有影响的技术论文。



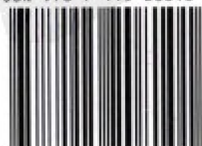
延伸阅读

- 《FPGA设计指南：器件、工具和流程》 Clive Maxfield 著 49.00元

本书译自原版*Design Recipes for FPGAs*，并由Elsevier授权出版。






ISBN 978-7-115-20810-1/TN

定价：39.00元

本书相关信息请访问：图灵网站 http://www.turingbook.com
读者/作者热线：(010) 51095186
反馈/投稿/推荐信箱：contact@turingbook.com

分类建议：电子电气/电路设计

人民邮电出版社网址 www.ptpress.com.cn